Agile

Desenvolvimento de software com entregas frequentes e foco no valor de negócio

ANDRÉ FARIA GOMES

Agradecimentos

Escrever este livro foi um grande desafio para mim, e passar por esse desafio foi um grande lembrete do quão valiosos são meus familiares, amigos, colegas de trabalho e de comunidade. Sem eles esse livro não teria se tornado realidade.

Agradeço à Editora Casa do Código nas pessoas de Paulo Silveira e Adriano Almeida pela oportunidade que me foi concedida e pela confiança para escrever sobre um assunto tão importante nos dias de hoje como o desenvolvimento ágil de sofware.

Agradeço à Bluesoft e a todos os seus colaboradores que sempre me apoiam e inspiram para buscar melhores práticas e abordagens no desenvolviemento de software e na gestão.

Em uma das vezes que foi entrevistado, Steve Jobs sugeriu que devemos nos expor às melhores coisas que seres humanos já fizeram (suas obras, seus trabalhos) e então tentar trazer essas coisas para o que você está fazendo. Bem, já faz algum tempo que eu venho tentando seguir esse conselho.

É por isso que eu agradeço também aqui a todos aqueles que desde o Manifesto Ágil vêm se dedicando para que possamos encontrar melhores maneiras de se desenvolver software.

Agradeço também à minha noiva Fernanda, que é minha maior fonte de inspiração e sempre me apoia em todos os meus desafios.

Finalmente, agradeço a você leitor, você é razão pela qual esse livro existe, sem você, esse trabalho não seria sequer necessário. Aproveite a leitura!

Quem sou eu?

Meu nome André Faria Gomes. Atualmente, sou sócio e diretor de produtos e tecnologia na Bluesoft em São Paulo e também Associated Trainer na Adaptworks.

Sou Bacharel em Sistemas de Informação pela FIAP, Black Belt em Lean Seis Sigma pela Fundação Vanzolini, e Management 3.0 Licensed Trainer.

O foco principal do meu trabalho é no desenvolvimento de software, atuando na liderança de equipes, no coaching de métodos ágeis, e no desenvolvimento de produtos para a Internet com diversas tecnologias e plataformas.

Minha carreira em TI começou em 1999 e desde então trabalhei com uma grande diversidade de tecnologias.

Desde 2007 venho aplicando métodos ágeis no dia a dia, e sempre buscando melhores formas de se lidar com os desafios da gestão e do desenvolvimento de software.

Atuo também como palestrante, tradutor, escritor e podcaster. Escrevo artigos para revistas e portais de TI, e mantenho meu blog andrefaria.com.

Prefácio

O ano era 2001 e eu estava prestes a abandonar a carreira de gerente de projetos de software. Eu não aguentava mais aquilo. Era o escopo que sempre mudava. O prazo e custo que sempre estouravam. O cliente que nunca sabia o que queria. A correria de fim de projeto. Fins de semana e madrugadas trabalhando. Conflitos. Prejuízo. E a eterna esperança de que "no próximo seria diferente". Não dava mais.

Naquele mesmo ano um amigo me emprestou um livro sobre uma tal FDD (Feature-Driven Development) e, após ler e ver sentido em muito do que estava ali, decidi me dar mais uma chance e tentar novamente, mas agora de uma forma diferente, afinal, pensei, se você não pode mudar uma situação, deve mudar sua atitude em relação a ela. Naquele momento, abrindo minha mente às possibilidades, abracei Agile - sem saber que aquilo era Agile - e mudei completamente o meu destino profissional. Depois do primeiro projeto conseguindo ter minha qualidade de vida e auto-estima profissional recuperadas, e vendo o sorriso no rosto do cliente, decidi mergulhar de cabeça neste mundo. Não haveria volta.

Hoje, depois do que vi na prática, nas trincheiras, por todos esses anos, eu afirmo a você: o resultado dos projetos de desenvolvimento de software que utilizam métodos ágeis é muito superior se comparado às técnicas mais tradicionais de gestão de projetos e engenharia de software. E quando eu falo em melhor resultado, não estou falando apenas de uma maior entrega de valor, tópico brilhantemente abordado neste livro, mas falo também de aspectos que vão desde a geração de produtos com qualidade técnica à construção de um melhor ambiente de trabalho. Estou certo de que em poucos anos nos lembraremos de Agile como um marco na nossa profissão, um marco para a área de tecnologia.

Mas, afinal de contas, o que é Agile? É uma metodologia? Um processo? Um conjunto de valores? Um manifesto? Ferramentas? Práticas? Um movimento? Bem, por incrível que pareça, esta é uma pergunta difícil de ser respondida. Uma das razões é porque Agile pode não ser nada do que citei, e, ao mesmo tempo, pode

compreender tudo aquilo. É muito difícil explicar Agile sem mostrar a prática. De fato, frequentemente cito que a forma correta de explicar o que é Agile deveria ser "Ei, venha aqui ver como estou fazendo!" E é neste ponto que destaco o valor de cada uma das páginas deste livro, elas mostram o Agile do "mundo real", infestado de pragmatismo e de preciosas anotações de quem valoriza sim uma boa teoria, mão não antes de pratica-la, de vê-la realmente funcionando.

Um livro de verdade sobre Agile não poderia ter capítulos cujos títulos fossem puramente relacionados a uma regra, artefato ou ferramenta de um ou outro método ágil. Um verdadeiro livro sobre Agile deveria manter o foco de seus capítulos na entrega de valor ao negócio, na otimização deste valor e na construção de um novo ambiente de trabalho, uma nova gestão. Um verdadeiro livro de Agile tiraria os holofotes dos famosos métodos ágeis, tais como Scrum, XP e Kanban, e os apresentaria apenas como um meio para se desenvolver da forma certa produtos que realmente agreguem valor a quem paga a conta: nossos clientes.

Sendo assim, não exito em afirmar que este é um verdadeiro livro de Agile. É o livro que você deve ler caso queira construir um novo e melhor caminho para a sua carreira na área de projetos de software.

Alexandre Magno

Agile Expert e Fundador da AdaptWorks

Sumário

Capítulo 1

Introdução à Métodos Ágeis

Na semana passada, entrevistei Joana, uma moça que havia passado 8 meses trabalhando em um projeto para um grande banco. Ela e sua equipe passaram todo esse tempo apenas fazendo levantamento de requisitos e documentando suas descobertas, até que descobriram que o problema que o software a ser desenvolvido se propunha a resolver já havia sido solucionado por uma ferramenta trazida de uma fusão com outro banco (há dois anos atrás) e já não era mais um problema. O projeto foi cancelado.

O resultado de 8 meses de trabalho de uma equipe, mais o investimento do tempo de diversos interessados que colaboraram com informações para o projeto se resumiu a um “bolo de documentação” que agora, provavelmente, nunca mais será sequer lida por ninguém. Joana estava frustrada com sua última experiência, e provavelmente quem quer que estivesse pagando por esse projeto também.

A experiência de Joana não é um caso isolado. Na verdade, muitos e muitos projetos tiveram e, infelizmente, ainda terão destinos semelhantes a esse. Mas por quê? Será que há uma forma melhor de gerenciar e desenvolver software que evite

tanto desperdício? A boa notícia é que há! Métodos ágeis são uma vacina contra o desperdício.

Por isso, nos últimos anos, os métodos ágeis vêm ganhando mais e mais popularidade, e grandes empresas como Google, Yahoo!, Microsoft, IBM, Cisco, Symantec e Siemens os têm utilizado [66]. Mas afinal o que os métodos ágeis trazem de diferente? O que tem despertado tanto interesse nesses grandes players do mercado de tecnologia? Para compreender melhor, vejamos como tudo começou.

No início da década de 90, no intuito de desburocratizar os processos de desenvolvimento de software, novas abordagens chamadas de "processos leves", como Scrum, Extreme Programming (XP), e Feature Driven Development (FDD), para citar algumas, começaram a emergir, mostrando-se mais bem sucedidas do que tentativas anteriores.

1.1 O Manifesto Ágil

Devido ao grande número de referências a esses processos leves, que emergiam como resposta aos constantes fracassos de projetos utilizando abordagens tradicionais, em fevereiro de 2001 um grupo de profissionais extraordinários do desenvolvimento de software reuniu-se em um Resort de Ski em Wasatch Range para discutir melhores maneiras de desenvolver software. Esse encontro deu origem ao manifesto ágil, uma declaração com os princípios que regem o desenvolvimento ágil [64]:

O Manifesto Ágil

Estamos descobrindo maneiras melhores de desenvolver software, fazendo-o nós mesmos e ajudando outros a fazê-lo. Através desse trabalho, passamos a valorizar:

- **Indivíduos e a interação entre eles** mais que processos e ferramentas;
- **Software em funcionamento** mais que documentação abrangente;
- **Colaboração com o cliente** mais que negociação contratual;
- **Responder a mudanças** mais que seguir um plano.

Mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda.

Além desses 4 valores, o manifesto ágil também é composto por 12 princípios:

- Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado.
- Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
- Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo.
- Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto.
- Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho.
- O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face.
- Software funcionando é a medida primária de progresso.

- Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente.
- Contínua atenção à excelência técnica e bom design aumenta a agilidade.
- Simplicidade - a arte de maximizar a quantidade de trabalho não realizado - é essencial.
- As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis.
- Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo.

Os profissionais que deram origem ao manifesto ágil foram Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland e Dave Thomas.

Ao longo desse livro nós estudaremos como esses valores e princípio são colocados em prática nas atividades e no dia-a-dia das equipes ágeis.

1.2 Métodos Ágeis

O Manifesto ágil é o embasamento filosófico de todos os métodos ágeis e diversos métodos de desenvolvimento de software estão alinhados a ele. A maioria deles se utiliza de ciclos curtos, que são chamados de iterações e normalmente têm duração de poucas semanas, dessa forma garantindo feedback frequente e respostas rápidas às mudanças.

Cada iteração contém todas as etapas necessárias para que se realize um incremento no produto, ou seja, no software: planejamento, análise, design, codificação, testes e documentação. Em métodos não ágeis, também conhecidos como métodos tradicionais, geralmente se encontra um processo em cascata em que todas as etapas citadas são executadas uma única vez e em sequência (ainda que idealmente, prevendo-se revisões incrementais de cada etapa, se necessário).

Scrum, Extreme Programming (XP), Crystal Clear e Feature Driven Development são exemplos de métodos ágeis. Cada um deles traz uma abordagem diferente que inclui diversos valores, práticas e reuniões. O Scrum, por exemplo, traz uma

abordagem mais voltada para a gestão, com maior foco nas reuniões, no planejamento e na melhoria contínua. Já o XP, trazem maior enfoque nas práticas técnicas. Ao decorrer do livro estudaremos melhor esses métodos e exploraremos algumas práticas ágeis.

A Cascata dos Métodos Tradicionais

"Não é o mais forte que sobrevive, nem o mais inteligente, mas o que melhor se adapta às mudanças."
– Charles Darwin

Em contra partida, os processos tradicionais ou em cascata (figura 1.1), que eram amplamente utilizados do mercado antes dos métodos ágeis, assumem que o desenvolvimento de software pode ser realizado através de uma sequência de atividades facilmente identificadas, previsíveis e repetíveis, mas diferente de outras engenharias, desenvolvimento de software requer criatividade, e geralmente envolve um alto nível de riscos e incertezas [38].

O processo cascata costuma ser realizado através de fases de análise de requisitos, projeto (design), implementação, testes, integração e manutenção, de forma que uma fase é iniciada somente quando a anterior termina. O termo foi originado em um artigo publicado em 1970 por W. W. Royce [52]. O próprio autor descreveu o método como arriscado e propenso a falhas.
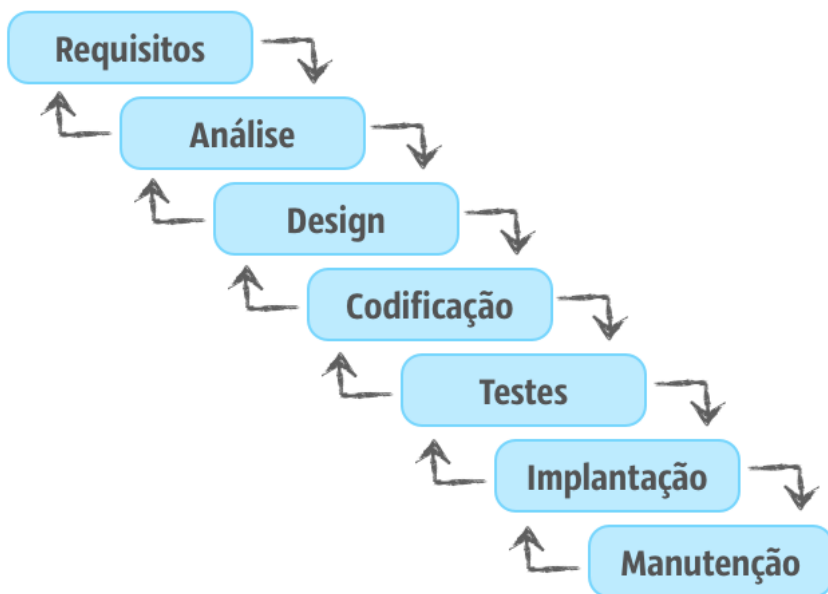
Figura 1.1: Processo em Cascata (Waterfall)

É claro que desenvolvimento ágil não é a única forma de se encarar o desenvolvimento de software, nem é a única maneira eficiente; como disse, Frederick Brooks em 1986, nenhuma tecnologia ou técnica de gestão resolve todos os problemas de todos os contextos. Ele resumiu essa ideia dizendo que "não há prata de bala" [15].

Métodos ágeis assumem imprevisibilidade natural do desenvolvimento de software, por isso, considera-se que o cliente também está aprendendo sobre o que precisa e, que a cada feedback pode mudar de ideia e alterar o escopo do projeto. Assume-se também que estimativas de esforço e tempo de desenvolvimento são, de fato, estimativas, e não devem ser tratadas como algo certo e sem margem de erro.

Por assumir a imprevisibilidade envolvida no desenvolvimento de software, métodos ágeis se baseiam nos ciclos de feedback constante permitindo que o cliente e a equipe de desenvolvimento possam adaptar-se às mudanças, aumentando assim as chances de sucesso do projeto.

Um fato interessante, que muitas vezes é deixado de lado nas discussões sobre métodos ágeis, é que o manifesto ágil não foi uma reação apenas aos métodos tradicionais e burocráticos, mas também foi uma reação aos métodos caóticos que resul-

tavam em produtos de baixa qualidade. Os métodos ágeis representam, justamente, um meio termo entre métodos estruturados demais e métodos sem estrutura, são o meio termo entre a ordem e o caos [8].

Com intuito de promover os métodos ágeis, foi instituída a Aliança Ágil (Agile Alliance), que apoiou e realizou uma série de eventos e conferências ao redor do mundo. Com o tempo, mais e mais empresas e pessoas foram adotando métodos ágeis e atualmente milhões de pessoas consideram-se praticantes desses métodos.

Métodos Prescritivos e Adaptativos

Métodos ágeis são adaptativos ao invés de prescritivos, por isso, incentivam a melhoria contínua (implicando em um constante estado de mudanças e transformação, visando alcançar um estado melhor) através de ciclos inspeção e adaptação. Esse é o motivo pelo qual métodos ágeis utilizam processos empíricos, em vez de prescritivos [36].

Enquanto processos empíricos são apropriados para domínios instáveis e com alto nível de mudanças, processos prescritivos são indicados para atividades ordenadas que podem ser alcanças através de uma sequência de passos.
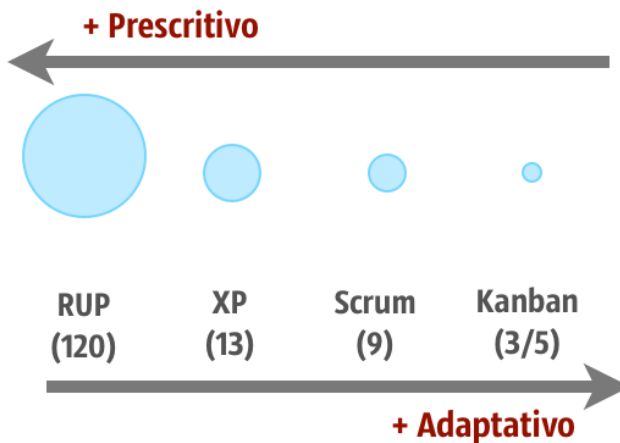
Figura 1.2: Métodos Mais e Menos Prescritivos

É por isso que métodos ágeis são menos explícitos em termos de papéis, atividades e artefatos do que métodos tradicionais (prescritivos). Para se ter uma ideia, o

RUP possui mais de 120 prescrições, o XP 13, o Scrum 9, e Kanban apenas 3 [34].

Quanto mais prescritivo um método for, mais específico para um determinado tipo de contexto ele será. Em contrapartida, quanto mais adaptativo, maior será sua aderência e flexibilidade para que seja otimizado com maior eficiência em diferentes contextos.

Em regra geral, ao utilizar um método prescritivo como RUP, por exemplo, você, possivelmente, precisará encontrará muito mais do que realmente precisa, e já com métodos menos prescritivos como Scrum, você precisará incluir tudo aquilo que estiver faltando para que o processo seja eficiente em seu contexto.

1.3 Compreendendo os Valores Ágeis

Conforme citado anteriormente, o Manifesto Ágil é formado por quatro valores fundamentais, que agora vamos explorar em mais detalhes.

O primeiro valor, que diz "**indivíduos e a interação entre eles mais que processos e ferramentas**", trata de entender que uma equipe é formada por pessoas e, que cada uma é diferente e única e possui pontos fortes e fracos, e não são vistas apenas "recursos" homogêneos e substituíveis.

O bom relacionamentos entre os membros da equipe é considerado crucial, e por isso, a agilidade do ambiente estimula o trabalho em equipe, a colaboração, e a comunicação constante. As equipes, geralmente, são formadas por pessoas com diferentes papéis, que se responsabilizam juntas pelo resultado do trabalho que realizam.

Processos são realizados por pessoas, e as ferramentas são utilizadas por pessoas. Se a interação entre elas não estiver fluída e bem equilibrada, provavelmente, a eficácia dos processos e ferramentas será comprometida. Por isso, nos métodos ágeis, as pessoas estão em primeiro lugar. Outro fator importante é que excelentes ferramentas e processos sem pessoas excelentes envolvidas, muito provavelmente, produzirão um resultado medíocre em vez de excelente [19].

Por outro lado, é importante ressaltar que as ferramentas também são importantes, apenas não são mais importantes do que as pessoas. Essa lógica vale para todos os valores do manifesto: o elemento da esquerda é mais importante do que o da direita, porém o da direita também é importante e relevante.

O segundo valor "**software em funcionamento mais que documentação abrangente**" é uma resposta a projetos tradicionais em que, por serem realizados por fases, costumava-se passar meses produzindo apenas documentação, que por si só, não agrega muito valor, ou, talvez nenhum valor ao cliente.

A natureza iterativa dos métodos ágeis permite que software em funcionamento seja entregue ao cliente em curtos períodos de tempo, dessa forma agregando valor maior em curto espaço de tempo. É claro que uma vez que a documentação é importante para o projeto, a cada entrega, ela poderá ser devidamente produzida e entregue junto com o software em funcionamento.

A "**colaboração com o cliente é mais valorizada do que negociação contratual**" justifica-se porque o objetivo da equipe ágil é entregar um produto que agregue valor, e para isso é preciso estar sempre pronto a adaptar-se às mudanças que, geralmente, ocorrem no mundo dos negócios, e consequentemente afetam uma ideia de escopo inicial que o cliente tinha do projeto a ser desenvolvido.

Contratos, geralmente, são necessários, mas muitos são protecionistas demais e procuram fechar o escopo do projeto desde o início, reduzindo assim as oportunidades de colaboração e descoberta junto com o cliente ao longo do processo de desenvolvimento, resultando em produtos que, muitas vezes, não atendem à necessidade de quem está pagando.

O importante é que o contrato matenha o máximo de opções abertas para que o projeto possa mudar na medida do necessário e dessa forma o projeto seja adaptável e, realmente, gere valor ao cliente.

Além disso, é essencial a colaboração e participação do cliente durante o desenvolvimento. Métodos ágeis procuram trazer o cliente para perto da equipe, o cliente faz parte do projeto e tem um papel muito importante para que o projeto seja bem-sucedido.

A única coisa constante é a mudança."
– Heráclito de Éfeso

Finalmente, "**responder a mudanças é mais importante do que seguir um plano**", diz respeito, essencialmente, à capacidade de adaptação que uma equipe ágil precisa possuir. Planejar é preciso, mas planos não precisam ser "escritos em pedras", eles podem ser apagados, corrigidos, refeitos.

A capacidade de adaptar-se em um mundo em constante mudança é uma qualidade essencial entregar projetos relevantes e bem-sucedidos.

1.4 Benefícios dos Métodos Ágeis

Uma das maiores motivações para a transição para métodos são os benefícios que são trazidos para a organização devido ao valor que é agregado ao cliente com qua-

lidade e velocidade [63]. Métodos ágeis ajudam organizações a responder mais rapidamente às necessidades do mercado, muitas vezes, resultando em grande vantagem competitiva.

Uma pesquisa realizada pela VersionOne.com em 2011 [65] envolveu mais de 6.000 pessoas e organizações dos mais variados perfis na industria de desenvolvimento de software. Ela nos mostra alguns dos principais benefícios obtidos por essas organizações após a transição par métodos ágeis. Os principais benefícios são:

- **Melhor Time-to-market e Maior Retorno sobre o Investimento**: Quanto mais cedo os clientes puderem começar a utilizar o produto, mais rápido receberá o retorno do valor investido no desenvolvimento do produto, seja através de lucros diretos gerados pelo produto ou através dos benefícios gerados pela utilização do produto na organização.

- **Maior Satisfação do Cliente e Melhor Gestão de Mudanças de Prioridades**: O planejamento iterativo permite que o cliente facilmente mude suas prioridades com impacto reduzido na produtividade da equipe porque planeja-se detalhamento apenas daquilo que está mais próximo de ser feito, evitando também desperdícios e custos desnecessários. A comunicação constante e a proximidade com o cliente frequentemente resulta também em um melhor alinhamento entre os objetivos de TI e com os objetivos de negócio da organização.

- **Melhor Visibilidade dos Projetos**: Faz parte da cultura ágil manter as informações do projeto visíveis e transparentes através de ferramentas como burndown-charts, e card walls (discutidas posteriormente), com as quais a equipe e gestão podem acompanhar dia a dia a evolução do projeto em relação às metas do projeto e das iterações;

- **Maior Produtividade**: Infelizmente, não há uma forma universalmente aceita de se medir produtividade de equipes de desenvolvimento de software. Muitos consideram essa tarefa até impossível ou algo subjetivo demais, mas na pesquisa supracitada, 75% dos participantes afirmaram ter alcançado melhor produtividade depois da transição para métodos ágeis;

- **Equipes mais Motivadas**: Métodos ágeis promovem ritmos sustentáveis de trabalho, são muitos os casos em que organizações reportaram uma diminuição significativa nas horas extras e madrugadas trabalhadas depois da transi-

ção para métodos ágeis [20]. A promoção do ritmo sustentável, de uma cultura de qualidade, de constante comunicação e trabalho em equipe são alguns dos fatores que contribuem para equipes mais motivadas e satisfeitas com seu ambiente de trabalho.

- **Melhor Disciplina na Engenharia e Melhor Qualidade Interna**: A utilização de práticas ágeis como redução de dívida técnica (melhorar a qualidade interna do produto), refactoring, desenvolvimento orientado a testes e programação em par, unido ao mindset de qualidade estimulado de pelos métodos ágeis, contribuem para a entrega de produtos com melhor mantenabilidade, extensibilidade e com menos defeitos.

- **Processo de Desenvolvimento Simplificado**: Os método ágeis são, em regra geral, menos prescritivos do que os métodos tradicionais, definem menos papéis e menos artefatos. São mais facilmente compreendidos pela equipe, e oferecem maior margem para otimização e adaptação para a maior eficiência no contexto da organização em que está sendo aplicado.

- **Redução de Risco**: O planejamento iterativo e as *releases* frequentes permitem que as prioridades do projeto sejam reajustadas constantemente.

  Métodos ágeis possibilitam que as incertezas do projeto, no nível dos requisitos, ou no nível técnico, sejam estimadas e distribuídas de forma inteligente nos *releases*, para manter o risco sempre balanceado. Além disso, as entregas em prazos curtos oferecem maior visibilidade da velocidade do time, oferecendo maior previsibilidade do prazo necessário para concluir o projeto.

- **Redução de Custos**: Equipes ágeis são menos propensas a desenvolver funcionalidades de baixa prioridade ou que se tornaram desnecessárias ao longo do tempo [20]. Abordagens não ágeis geralmente tendem a cair nesse problema devido ao grande espaço de tempo entre o levantamento dos requisitos e entrega do produto.

1.5 Agregando Mais Valor com Scrum
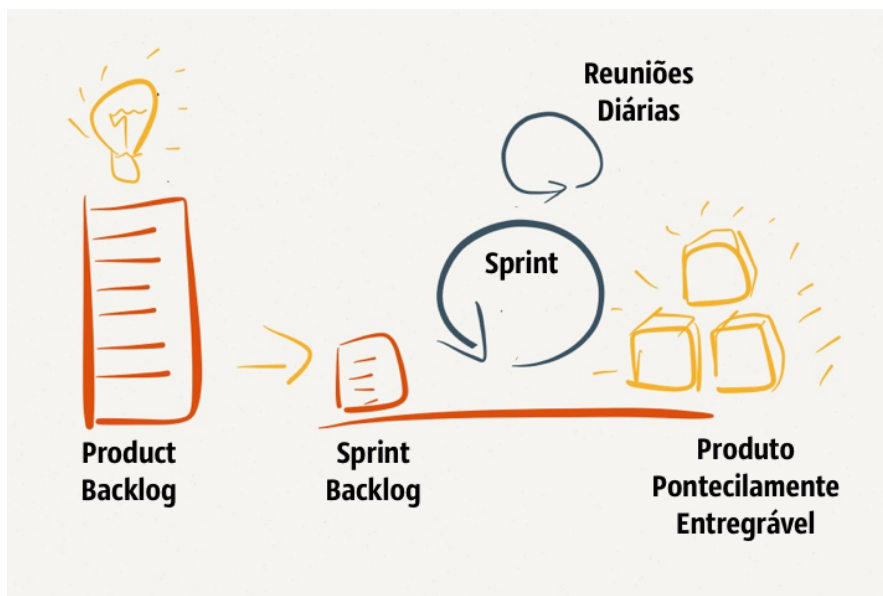
Figura 1.3: Scrum

Scrum é um dos métodos ágeis mais populares na atualidade e tem foco maior nos aspectos gerenciais do desenvolvimento de software. Nele, cada iteração é chamada de sprint. Geralmente cada sprint tem um período de mês que pode variar de poucos dias a algumas semanas. As pessoas envolvidas no processo de desenvolvimento são dividas no Scrum em três papéis principais: o Scrum Master, o *Product Owner* (dono do produto) e a equipe.

O *Product Owner* é o maior interessado no software, é aquele que teve (ou que representa quem teve) a necessidade do produto, e por isso tem a visão do produto. Define o que deve ser feito, e prioriza as funcionalidades a serem desenvolvidas, mantendo-as em um artefato chamado de *product backlog*, que nada mais é do que uma lista de todas as funcionalidades que devem ser implementadas, ordenadas por prioridade.

É também responsabilidade do *Product Owner* passar toda a informação de negócio que for necessária para que a equipe possa transformar suas ideias em software.

A Equipe é composta por um número que varia de cinco a nove pessoas e é *cross-funcional*, isto é, há pessoas dos mais variados perfis, como testadores, desenvolve-

dores, designers, analistas de negócio etc. integrando a equipe. O principal objetivo dela é implementar as funcionalidades que foram selecionadas para serem desenvolvidas na iteração e entregá-las em funcionamento ao final desse período. No capítulo 6 abordaremos mais a fundo as vantagens de se trabalhar com equipes pequenas.

O Scrum Master, também chamado de facilitador, é responsável por manter o processo em funcionamento, assegurando que todas as regras estão sendo aplicadas, e remover os impedimentos da equipe, isto é, resolver qualquer problema que possa atrapalhar o progresso do desenvolvimento, garantindo assim que o objetivo da iteração seja atingido.

É importante ressaltar que o Scrum Master não atua como um gerente ou chefe da equipe porque ela é auto-organizável. O Scrum Master não determina o que cada membro da equipe deve ou não fazer: a equipe se compromete com a entrega das funcionalidades e, então, se auto-organiza, definindo por quem e em qual momento as tarefas serão realizadas.

Toda sprint começa com uma reunião de planejamento, que é dividida em duas partes. A primeira delas tem um enfoque mais estratégico, na qual se decide o que será feito, isto é, quais funcionalidades serão implementadas e define-se uma meta para a Sprint. Para isso, o *Product Owner* apresenta os itens do *product backlog*, e a equipe obtém informações suficientes sobre cada um dos itens, dessa forma, é possível fornecer uma estimativa que expresse um tamanho ou um número de horas para o trabalho e assim seja definido quantas e quais tarefas poderão ser desenvolvidas no sprint (de acordo com a velocidade da equipe que é calculada através de dados de sprints passados).

Depois de definidas quais tarefas serão desenvolvidas no sprint, a equipe utiliza a segunda parte da reunião, que tem um enfoque mais tático, para decidir como serão feitas as tarefas. Então se analisa tarefa por tarefa com mais detalhes, mais informações de negócio são apresentadas e é possível tomar diversas decisões de negócio e decisões técnicas.

Ao fim da reunião de planejamento a equipe começa a trabalhar nas tarefas, respeitando as prioridades: fazendo sempre primeiro as tarefas mais importantes, que representam maior valor para o produto de acordo com o *Product Owner*. Todos os dias é realizada uma reunião para que a equipe converse sobre o andamento do sprint. Posteriormente apresentaremos em mais detalhes as reuniões.

Ao longo da Sprint a equipe mantém sempre em mente a sua meta e quando o andamento das coisas foge do que foi previsto, ela pode negociar o escopo com o *Product Owner*, de forma que não comprometa a meta [59].

Timeboxes

O Scrum reforça a ideia de que todas as cerimônias do time precisam ter um *timebox* (tempo definido) e que é muito importante que esses tempos sejam respeitados para se manter um ritmo sustentável. Se as *Sprints*, por exemplo, tem duração de duas semanas, é importante que não sejam extendidas. Se a reunião diária dura 15 minutos, é importante que a reunião não seja extendida. Os *timeboxes* das outras reuniões variam de acordo o *timebox* da *Sprint* e e podem variar também de acordo com cada contexto da equipe.

Os tempos sugeridos para um Sprint de 30 dias são [59]:

- Reunião Planejamento: 8 horas.
- Reunião de Revisão (Demonstração): 4 horas.
- Reunião de Retrospectiva: 3 horas.

Ao fim do sprint, mais duas reuniões acontecem: a reunião de revisão e a reunião de retrospectiva. Na reunião de revisão a equipe apresenta ao *Product Owner* o trabalho que foi desenvolvido durante o sprint, para que ele possa oferecer feedback, e aprovar ou não tudo o que foi produzido. Na reunião de retrospectiva os principais acontecimentos do sprint são apresentados e discute-se sobre as lições aprendidas e melhorias que podem ser aplicadas ao processo.

1.6 Excelência Técnica com XP

O método ágil *Extreme Programming* (em português Programação Extrema), mais conhecida simplesmente como XP, foi criado e inicialmente divulgado por Kent Beck nos anos 90, e é um dos métodos ágeis que melhor cobre aspectos técnicos do desenvolvimento de software como codificação, design e testes, veremos mais detalhes a seguir.

Segundo o XP, durante o processo de desenvolvimento há quatro atividades básicas a serem executadas: Ouvir, Desenhar, Codificar e Testar.

É preciso ouvir porque desenvolvedores nem sempre possuem o conhecimento

de negócio necessário para se construir o software. O *Planning Game* é uma reunião que acontece uma vez a cada iteração, em que o principal objetivo é decidir quais funcionalidades serão desenvolvidas na iteração e aprender mais sobre elas. O cliente escreve **histórias de usuário** em cartões que representam a necessidade de funcionalidade a ser desenvolvida, e explica para os desenvolvedores tudo o que for preciso para que eles possam implementá-la.

Um bom *design* é uma excelente ferramenta para que todos possam compreender melhor os sistemas complexos, suas estruturas, dependências e regras de negócio. O principal objetivo é manter o *design* sempre simples, evitando aumentar a complexidade com a criação de funcionalidades que não sejam realmente necessárias.

Essa é a máxima do principio YAGNI (*You aren't gonna need it* – Você não vai precisar disto) que basicamente diz que devemos fazer as coisas somente no momento em que realmente precisarmos delas: muitas vezes antecipamos o desenvolvimento antes mesmo que surja a necessidade real, assumindo que um dia ela chegará, porém, essa necessidade pode nunca chegar.

Ao manter o design simples e fazer somente aquilo que for necessário, você poupará tempo, porque deixará de escrever código que não precisa e terá mais tempo para se concentrar na qualidade do que realmente é necessário e que vai agregar valor para a demanda real e atual do cliente.

Codificar é uma atividade essencial para o desenvolvimento de software, afinal de contas, sem código não existe software algum. Nessa etapa, é extremante importante que o cliente continue acessível para que possa oferecer feedback e responder às diversas dúvidas que surgem durante o desenvolvimento.

Com XP, os testes de unidade são escritos antes do código de produção (discutiremos em detalhes mais à frente); todo o código fonte que será executado em produção é **desenvolvido em pares**; a integração do código fonte é realizada frequentemente através da prática de **integração contínua**; e o **código fonte é coletivo**, ou seja, pertence a todos os membros da equipe, e deve ser escrito de acordo com os padrões definidos pelo próprio time.

Testar é uma atividade de extrema importância para garantir a qualidade do software e não é algo opcional com XP, ao contrário, todo código deve possuir testes de unidade, e todos os testes devem ser executados com sucesso antes que uma entrega seja feita. Quando um defeito é encontrado, cria-se um teste de unidade para assegurar que esse mesmo defeito jamais volte a acontecer. No XP os testes são escritos antes do código de produção através de **TDD** (*Test Driven Development*, ou Desenvolvimento Guiado por Testes).

Devido ao foco técnico de XP e ao foco mais gerencial do Scrum, muitas empresas os combinam para obter um processo mais completo, visando um processo mais eficiente. Estudaremos mais a frente cada uma das práticas citadas.

1.7 Fluxo Contínuo com Kanban
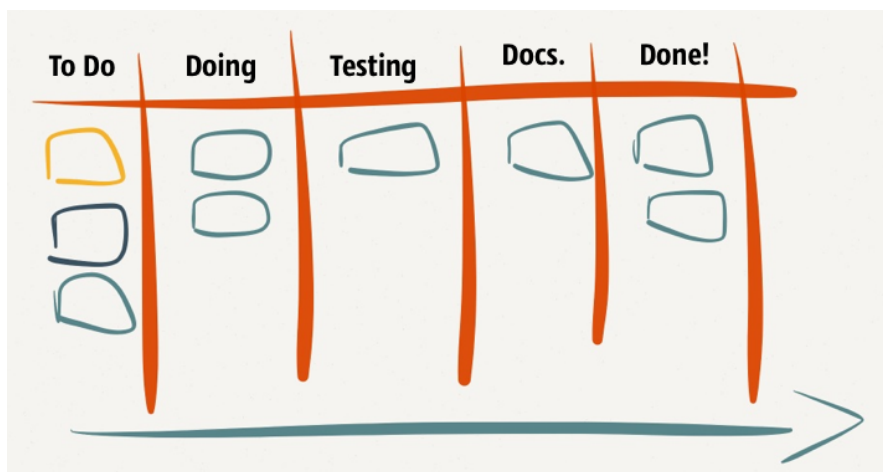
Figura 1.4: Kanban

Kanban é um método ágil que, diferentemente da maioria dos métodos ágeis, não possui iterações. Ao invés disso, desacopla o planejamento, priorização, desenvolvimento e entrega, de forma que cada uma dessas atividades possa ter sua própria cadência para melhor se ajustar à realidade e necessidade que o processo demanda.

Cadências são repetições que se sucedem em intervalos regulares. Este é um conceito do método Kanban que determina o ritmo de um determinado tipo de evento. Priorização, entregas, retrospectivas, e qualquer evento recorrente podem ter sua própria cadência.

Métodos ágeis, no geral, controlam cadências através das iterações que geralmente duram de uma a quatro semanas. Dentro de cada iteração, realiza-se planejamento, desenvolvimento, testes, revisões, retrospectivas etc. Tudo isso dentro de um período pré-definido.

Já com Kanban o processo é um fluxo contínuo e não é preciso que todo o trabalho planejado esteja concluído para que uma entrega seja realizada. No momento

da entrega, algumas tarefas estarão prontas e outras em progresso; as que estiverem prontas farão parte da entrega, e as que não estiverem ficarão para a próxima.

Em meados de 2007, Rick Garber e David J. Anderson apresentaram nas conferências o "Lean New Product Development" e "Agile 2007", os resultados preliminares do uso de Kanban na Corbis, uma empresa que vende imagens e clips de vídeo fundada por Bill Gates, da Microsoft. Desde então, o Kanban vem ganhando mais e mais força na comunidade de desenvolvimento de software e mais empresas passaram a adotá-lo.

Kanban está bastante relacionado com o conceito de Pull Systems (sistemas de produção puxados), do *Toyota Production System* (TPS). Tradicionalmente, antes da Toyota, a grande maioria das indústrias utilizava o conceito de Push Systems (sistemas de produção empurrada).

Um sistema de produção empurrada caracteriza-se quando a primeira operação do processo recebe uma ordem de produção e a executa, empurrando o resultado da operação atual para a operação seguinte. Em sistemas empurrados, não há ligação direta entre o que é produzido e a demanda do cliente, ou seja, pode-se dizer que o fato de um item ter sido produzido não significa que um outro item tenha sido vendido.

Já um sistema de produção puxada exige que cada operação do processo seja alimentada pela demanda da etapa anterior, estabelecendo uma ligação direta entre o consumo real do cliente e a quantidade produzida. Dessa forma, o fato de um item ter sido vendido, geraria a demanda para a fabricação de outro. Em uma abordagem de desenvolvimento de software, poderíamos dizer que a demanda para se trabalhar em uma nova funcionalidade seria gerada quando alguma outra fosse entregue.

Kanban é uma palavra japonesa que significa cartão sinalizador. É utilizado no Sistema Toyota de Produção (TPS) e também por diversas empresas que aderiram à filosofia Lean para representar, em um sistema puxado de produção, um sinal para que se puxe mais trabalho, ou seja, para que o processo seja alimentado.

Kanban é dos métodos de desenvolvimento de software menos prescritivos, que, de acordo com Henrik Kniberg, possui apenas três prescrições [34]:

1) Visualizar o fluxo de trabalho (workflow);

2) Limitar o trabalho em progresso;

3) Gerenciar e medir o fluxo.

Kanban incentiva a adaptação baseada em contexto. Deste modo, parte-se da premissa que não existe uma única solução que resolva todos os problemas de todos os diferentes contextos, e por isso há a necessidade de adaptação, em que cada processo deve se adequar às necessidades do contexto a que pertence.

Kanban também recomenda que se meça o **lead time**, tempo médio para se completar um item, de forma que o processo deve ser continuamente otimizado para tornar esse tempo cada vez menor e mais previsível.

1.8 Qual é o Melhor Método?

Kniberg define ferramenta como "qualquer coisa que possa ser usada para atingir uma tarefa ou objetivo" , e processo como "a forma que se trabalha" [34]. Para ele, métodos como Scrum, XP e Kanban são ferramentas para processos, e questionar qual é o melhor deles é o mesmo que comparar se um garfo é melhor do que uma faca, ou seja, isso depende do contexto e do objetivo em questão.

Nenhum método é completo e nenhum é perfeito, por isso, sinta-se livre para combinar as ferramentas e aproveitar o que funcionar melhor para seu contexto, ou seja, considerando a cultura da sua organização, as habilidades da sua equipe e as restrições de seu projeto. É comum, por exemplo, que se adote práticas de engenharia de XP, como programação em par e integração contínua, em times que utilizam Scrum.

Todo projeto de desenvolvimento de software é diferente, possui pessoas com habilidades diferentes, níveis de experiência diferentes, orçamento, prazo, escopo e riscos diferentes. Os valores da organização e suas metas também são específicos, e com toda essa variação, sem dúvida, as restrições também serão diferentes. Portanto, você deve procurar compreendê-las de forma a explorá-las da melhor maneira possível visando atingir bons resultados.

Assim como as práticas, os papéis em uma equipe ágil, podem variar de acordo com o método utilizado e contexto da equipe. Fique atento para utilizar apenas papéis que de fato agreguem valor, e para não incluir papéis conflitantes.

Cada novo método que você estuda e conhece é mais uma ferramenta para sua caixa de ferramentas, a pergunta certa não é "qual é o melhor método", mas "qual o método mais adequado para esse contexto", e então, munido de sua caixa de ferramentas, você terá condições de escolher a melhor abordagem para cada desafio.

1.9 E agora, o que eu faço amanhã?

Revise os quatro valores ágeis: você acredita que sua organização reflete esse valores? Se não, o que poderia ser feito diferente para que isso melhorasse?

Reflita sobre seu projeto atual: Você e seus colegas estão de fato trabalhando colaborativamente em equipe? Estão entregando software em funcionamento com numa frequência adequada? Como vocês lidam com as mudanças de prioridade de negócio do cliente?

Quais são os benefícios que você espera alcançar com a utilização de métodos ágeis em suas equipe? O que métodos ágeis podem trazer de positivo para seu cliente? Como podem beneficiar sua equipe? Verifique cada um dos benefícios listados neste capítulo e pense se estão em alinhamento com o que você espera melhorar em sua organização.

Pense nos diferentes papéis e práticas de sua equipe. Todos eles de fato estão agregando valor ao projeto?

CAPÍTULO 2

Fluência Ágil

Em um experimento científico [58], um grupo de cientistas colocou cinco macacos numa jaula. No meio da jaula, havia uma escada e sobre ela, um cacho de bananas. Quando um macaco subia na escada para pegar as bananas, os cientistas jogavam um jato de água fria nos outros macacos que estavam no chão.

Depois de certo tempo, quando um macaco tentava subir na escada, os outros batiam nele (com medo de tomar outro jato). Depois de algum tempo, nenhum macaco subia mais a escada, apesar da tentação das bananas.

Os cientistas, então, trocaram um dos macacos por outro novo. Quando ele tentou subir a escada para pegar as bananas, também apanhou dos outros. Outro macaco foi substituído e o mesmo episódio se repetiu por diversas vezes, e inclusive os novos macacos aprenderam a bater nos outros que tentavam pegar as bananas.

Depois de algum tempo, todos os macacos foram substituídos, porém o hábito permaneceu, e mesmos os macacos que nunca viram o jato d'água continuavam batendo naqueles que tentavam subir para pegar bananas.

Se fosse possível perguntar a algum deles porque eles batiam em quem tentasse

subir a escada, com certeza a resposta seria algo do tipo: "Não sei, mas as coisas sempre foram assim por aqui".

Você já viu alguma coisa desse tipo acontecer nas empresas em que trabalhou? Alguma regra que já não faz mais sentido, mas que, porém, continua sendo utilizada por todos, mesmo sem que saiba o motivo? Infelizmente, isso não é tão incomum, e inclusive equipes ágeis podem cair nesse engano.

Métodos ágeis ganharam bastante popularidade e, atualmente, vêm sendo utilizados por grande parte das organizações que desenvolvem software, mas essa popularidade também trouxe alguns problemas.

Muitos passaram a se interessar por métodos ágeis apenas porque grandes players do mercado estão utilizando, porém não buscaram entender a essência (colaboração, agregar valor de negócio, entregas frequentes etc.), e ficaram apenas na forma (quadros na parede, estimativas com cartas, backlogs, etc.).

Diana Larsey e James Shore [57] afirmaram que muitos líderes de organizações ainda se queixam de que não estão conseguindo alcançar os benefícios que esperam com a adoção de métodos ágeis e, por isso, desenvolveram um modelo de fluência ágil para ajudar as organizações a alcançarem esses benefícios com mais eficiência.

Para Larsey e Shore, a fluência ágil diz respeito à maneira com que a equipe responde quando está sob pressão: "Qualquer um pode seguir um conjunto de práticas quando há tempo para focar em uma sala de aula, mas a verdadeira fluência requer habilidade e prática frequente que persiste mesmo quando sua mente está distraída com outras coisas, é uma questão de hábitos".

Essa fluência ágil está relacionada não apenas com a capacitação dos membros da equipe, mas também com as estruturas de gestão, com os relacionamentos, a cultura organizacional e práticas da equipe.

Para conquistar a fluência ágil, uma equipe deverá passar por 4 estágios distintos:

1) Foco no Valor (Atuar na Cultura na Equipe).

2) Entrega de Valor (Atuar nas Habilidades da Equipe).

3) Otimização de Valor (Atuar na Estrutura Organizacional).

4) Otimização Sistêmica (Atuar na Cultural Organizacional).

Ao longo deste livro, nós exploraremos o que você pode fazer para ajudar sua equipe e sua organização a trilhar esse caminho que os levará a um melhor aproveitamento dos benefícios que os métodos ágeis podem oferecer. E para que não

façamos como os macacos, nós vamos buscar entender não apenas a forma das práticas, mas também a essência de cada uma delas.

2.1 Evolução e Maturidade de uma Equipe Ágil

"Unir-se é um bom começo, manter a união é um progresso, e trabalhar em conjunto é a vitória."
– Henry Ford

Sempre que uma nova equipe é formada, naturalmente, um processo de maturidade acontece [51]. É notável que um equipe que acabou de se formar não trabalha com a mesma produtividade e dinamismo que uma equipe cujas pessoas já tiveram algum tempo para se conhecer, os seus pontos fortes e suas dificuldades.

Métodos ágeis requerem verdadeiras equipes, não apenas grupos [53]. E verdadeiras equipes ágeis são formadas de pessoas, com uma diversidade de habilidades, que trabalham colaborativamente com uma visão compartilhada para atingir determinados objetivos.

As pessoas precisam trabalhar algum tempo juntas e passar por todos esses estágios, por isso não se deve mover pessoas de uma equipe para outra com alta frequência, como se fossem peças de xadrez, essa instabilidade toda pode destruir a integridade do time. Isso não significa que de vez em quanto não seja interessante que as pessoas mudem de equipe para que possam diversificar suas experiências, mas é preciso encontrar um ponto de equilíbrio saudável. Ambos os extremos são ineficientes.

No artigo *"Your Path to Agile Fluency*", Diana Larsey e James Shore [57] afirmam que o "*turnover* é a principal causa da perda de fluência em uma equipe ágil, e que quando uma equipe ganha ou perde membros, enfrenta problemas para sustentar aquilo que já havia sido aprendido."

Em 1977, Bruce Tuckman, criador da teoria das dinâmicas de grupos, publicou uma artigo chamado "Estágios de Tuckman" que descreve esse processo de maturidade em quatro estágios: Formação, Confronto, Normatização e Performance (veja na figura 2.1.
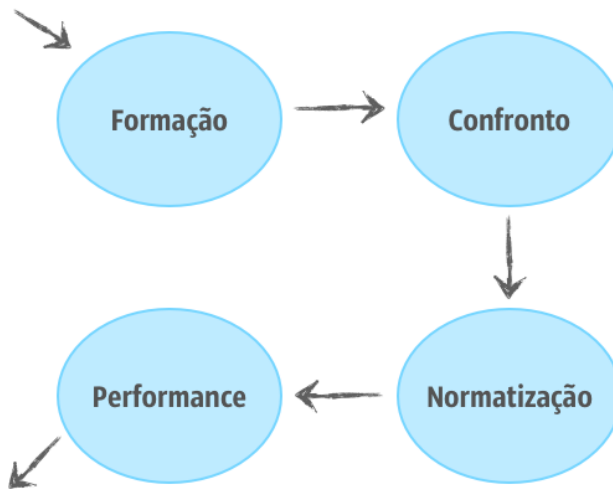
Figura 2.1: Estágios de Tuckman

Cada um dos estágios citados anteriormente possui algumas características próprias:

1. Formação: Este é o primeiro estágio pelo qual uma equipe recém formada passará, é o momento em que os membros da equipe começam a conhecer uns aos outros, procuram entender quais são as habilidade de cada um, e as tendências de comportamento de cada pessoa. Nesse estágio também se estabelecem os primeiros objetivos coletivos e completam-se algumas tarefas, mas a ênfase das pessoas ainda é mais individual do que coletiva.

2. Confronto: Conflitos aparecem à medida que diferentes ideias para atingir objetivos vão surgindo e as metas coletivas identificadas no estágio de formação são questionadas. Alguns membros podem se abalar por essa situação. Uma liderança forte é extremamente importante para que a equipe passe por este estágio.

3. Normatização: É neste estágio que as pessoas realmente passam a priorizar os objetivos coletivos da equipe sobre os objetivos individuais de cada membro, por isso, a equipe consegue entrar em consenso mais facilmente.

4.Performance: A equipe já possui todas as competências necessárias para atingir seus objetivos, todos estão comprometidos e motivados, e atuam com verdadeiro espírito de equipe, pensando em termos de "nós" em vez de "eu" ou "eles". Conflitos são facilmente solucionados.

Maturidade Ágil

Além de uma equipe unida e forte que saiba trabalhar colaborativamente, é preciso aprender a se trabalhar com métodos ágeis, e amadurecer na utilização das práticas ágeis.

A arte marcial Aikido, utiliza um método de aprendizagem desenvolvido no Japão a mais de 4 séculos atrás chamado **Shu-Ha-Ri** [17] (figura 2.2) que consiste em três níveis de habilidade que podem ser utilizados como referência para qualquer nova prática, habilidade ou competência a ser aprendida, inclusive métodos ágeis. Os três estágios são:

Figura 2.2: Shu Ra Hi

1. Shu: Representa o primeiro estágio da aprendizagem de uma habilidade, em que é necessário se construir a base, a fundação do que se está aprendendo.

O aprendiz busca observar, copiar e imitar o que alguém que já tem experiência faz, geralmente, um mestre. Neste estágio é mais eficiente seguir regras do que tentar improvisar ou modificar técnicas.

Uma equipe que está começando a trabalhar com métodos ágeis, certamente, por algum tempo estará nesse estágio. É um momento importante para seguir as regras do método numa abordagem "**by-the-book**", ou seja, sem alterações ou improvisos. Se uma equipe está utilizando Scrum, neste estágio, por exemplo, é importante que todos os papéis sejam assumidos, e que todas as reuniões sejam realizadas de acordo com as "regras". Esse é um momento em que pode ser importante procurar ajuda de alguém que tenha experiência com métodos ágeis.

2. Ha: Esse é o segundo estágio, em que o estudante já começa a pensar mais profundamente sobre os porquês das coisas, quebra algumas regras, e rompe um pouco

com a tradição. Agora, já é possível refletir sobre o que foi aprendido e compreender melhor os princípios em vez de apenas repetir ou seguir regras.

É nesse estágio que algumas adaptações ao contexto podem começar surgir, novas práticas são inseridas, e *insights* de experiências passadas servem como base para tomadas de decisão. Agora, mesmo que as coisas mudem um pouco em sua forma (práticas, papéis, reuniões), a equipe estará pronta para manter a essência da agilidade (entregas frequentes, respeito às pessoas, comprometimento, colaboração, comunicação etc.).

3. Ri: Neste último estágio, a prática já se torna parte de quem está aprendendo e, apesar de ainda seguir algumas regras (mesmo aquelas que foram quebras e reinventadas) [4], tudo parece ser natural, e o aprendizado vem da prática e da experiência.

É importante notar que uma equipe pode estar em estágios diferentes dependendo da referência, por exemplo, um time pode estar no estágio *Ri* em se falando de entregas frequentes, mas pode estar no estágio *Shu* em relação à programação em par.

Uma lição importante deste modelo é que há um momento certo de maturidade para se quebrar as regras, adaptar e improvisar. Um aluno iniciante (no estágio *Shu*) de artes marciais que tentar mudar os movimentos ensinados pelo mestre poderá facilmente fazer algo errado e se machucar, já um aluno intermediário (no estágio *Ha*), provavelmente, estará mais preparado para encontrar seu próprio estilo sem correr altos riscos.

O mesmo ocorre com uma equipe que está utilizando métodos ágeis, se num momento precoce tentar adaptar ou quebrar as regras, antes de ter compreendido mais profundamente os princípios, corre o risco de estar fazendo alguma outra coisa qualquer e as continuar chamando de métodos ágeis.

2.2 Ordem, Caos e Complexidade

Assim como há muitos tons de cinza entre o preto e o branco, entre o caos (desordem) e a ordem, há também alguns níveis [8]. Quando tentamos trazer nossas organizações para o extremo da ordem, para que possamos tornar as coisas mais previsíveis e manter tudo sob controle, criamos uma série regras, e enrijecemos o sistema, tornando-o pouco adaptativo e burocrático. Por outro lado, quando não há nenhuma regra, nenhuma restrição, nenhum líder, o sistema se encontra no que chamamos de caos sem nenhuma previsibilidade e você não consegue ter a menor ideia do que está acontecendo ou para aonde as coisas estão caminhando.

Para ilustrar a diferença entre a Ordem e o Caos, imagine que você está indo com sua família passar o final de semana na praia. Você tem dois filhos pequenos e danados, um menino, e uma menina. Seu cônjuge foi fazer um caminhada e você ficou responsável pelas crianças. Se você optar pelo Caos extremo, não dirá nada para as crianças e as deixará completamente livres para fazerem aquilo que quiserem, não as advertirá, não haverá limites, nem restrições, nem regras. Liberdade total, tudo é possível! Nem é preciso falar que a probabilidade de uma criança se afogar e a outra entrar no meio de um jogo de futebol e tomar uma bolada é bem alta.

Agora, se você optar pela ordem extrema, dirá a seus filhos:

Sentem-se aqui na minha frente, brinquem de fazer castelo com 500 metros de altura e duas torres, usem o balde e as pazinhas, não saiam daqui enquanto eu não autorizar, não chorem, não gritem, não levantem, e (imagine e inclua mais umas 50 regras aqui).
–

Num contexto totalmente ordenado, todas as variáveis devem ser isoladas, e por isso é possível prever acontecimentos, porém, note que ambos os extremos são exagerados. No primeiro, as crianças acabam se machucando por excesso de liberdade e negligência, no segundo exemplo, as crianças mal podem se divertir pois precisam seguir ordens restritas e não têm liberdade alguma para se auto-organizar ou expressar sua individualidade e criatividade.

O meio termo entre esses extremos, seria dar às crianças algumas restrições básicas para certificar-se de que elas poderão brincar sem que nada de mal ocorra a elas. Você diria algo do tipo: - "Crianças podem brincar do que quiserem, e fiquem onde eu possa vê-los, e não deixem que a água ultrapasse a cintura de vocês."

Agora pense em uma equipe de desenvolvimento de software que não possui nenhuma regra comum entre os membros da equipe, cada um escreve código da maneira que acha melhor, uns escrevem testes outros não, alguns utilizam certas convenções, outros utilizam outras, cada um trabalha em sua linguagem de programação preferida, alguns trabalham no escritório outros de casa, cada um define sua frequência de integração do código.

Dá pra imaginar que esse cenário seria um bagunça, não é? Por outro lado uma equipe que está no extremo da ordem possui regras para tudo, não tem nenhuma liberdade para expressar sua criatividade ou para solucionar os problemas, todos os horários estão definidos, as tecnologias, as linguagens de programação, o que se pode fazer, o que não se pode fazer. Há pouca ou nenhuma variação.

Os métodos ágeis são uma resposta ao caos e à ordem, e propõem um cenário que está justamente no meio termo: a complexidade [8] (veja na figura 2.3. Muitos autores chamam isso de a "beira do caos". Você só precisa de um pouco ordem para que o sistema possa se auto-organizar para atingir um determinado objetivo. Não mais do que isso. E essa "ordem" geralmente está presente nos métodos ágeis na forma de **restrições**.
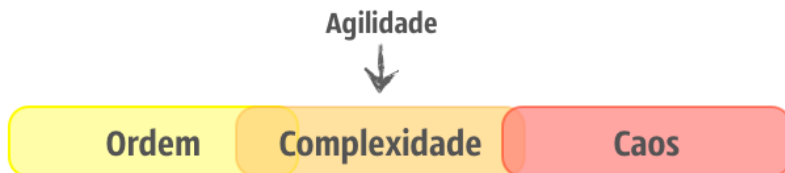
Figura 2.3: Complexidade

"Toda organização é um sistema complexo adaptativo, é como um jogo em que as regras são mudadas ao longo do curso, e pelos próprios participantes."
– Jurgen Appelo

Pense no Scrum, como um exemplo. A equipe precisa possuir de 5 a 9 membros. Todos os dias o time deve fazer uma reunião diária de até 15 minutos. Toda iteração tem como resultado um potencial incremento no produto etc. Essas regras e restrições asseguram que a equipe trabalhe em um estágio intermediário entre a ordem e o caos. Tudo aquilo que fica implícito é espaço para que ela possa se auto-organizar e adaptar-se para realizar o melhor trabalho possível no contexto em que está inserida.

Em uma abordagem ágil, o trabalho do gestor não é criar regras na organização, mas certificar-se de que as pessoas podem criar suas próprias regras juntas, e é justamente esse esforço conjunto que permite que o sistema alcance "a beira do caos" [8]. Nessa visão a gestão ocupa-se apenas com algumas restrições de alto nível. O resto pode ser definido pela própria equipe, e é nesse cenário que a auto-organização acontece, porque ela tem liberdade para criar suas próprias regras e tomar decisões. Quando todas as decisões vêm de cima para baixo, não há auto-organização.

Auto-organização requer empoderamento, e empoderamento requer confiança. A gestão precisa fazer um investimento de autonomia para que equipe possa tomar decisões e se organizar da forma mais eficiente para atingir os objetivos da organização.

Outro ponto importante, é que **a auto-organização por si só, não é boa, nem ruim e pode levar a qualquer resultado**. Por isso é essencial que a equipe se auto-organize em torno de um objetivo, de metas importantes para o sucesso. É como se a gestão apontasse aonde a organização quer chegar e qual é a direção a seguir, mas a equipe tivesse autonomia suficiente para decidir qual é a melhor forma de se chegar até lá.

2.3 E agora, o que eu faço amanhã?

Faça um levantamento do *turnover* da sua equipe. Quantas pessoas deixaram a equipe nos últimos seis meses, e quantas novas pessoas foram recebidas? Reflita sobre o impacto dessas mudanças na produtividade da sua equipe. Vocês rotacionam os membros das equipes de tempos em tempos? Isso foi bom ou ruim? Por quê? Qual tem sido o impacto dessas mudanças na performance da equipe?

Pense em qual dos estágios Tuckman sua equipe se encontra no seu ponto de vista em relação à agilidade. O que pode ser feito para vencer os desafios atuais e seguir para o próximo estágio?

Converse com sua equipe sobre os seus objetivos. Vocês sabem qual é o papel de vocês na organização? Caso estes objetivos não estejam claros, procure compreendê-los, conversar com a gestão, e torná-los explícitos para que toda a equipe saiba para onde estão indo e aonde devem chegar.

A equipe tem liberdade para decidir qual a melhor forma de atingir seus objetivos, ou as decisões são impostas pela gestão? Discuta com sua equipe sobre como vocês podem ganhar mais confiança da gestão para criar um ambiente mais favorável à auto-organização.

Pense sobre as quais são as regras da sua equipe. São realmente necessárias? Será que alguma delas foi útil no passado mas agora já perdeu o sentido?

Capítulo 3

Foco em Valor para o Negócio

Não é incomum ouvirmos histórias de times que passaram anos construindo um produto que ninguém jamais usou, ou então de equipes que passaram meses otimizando um produto para que ele tivesse máxima perfomance e escalabilidade, construindo sempre com tecnologia de ponta, mas a primeira frase do cliente ao ver o produto funcionando é "Não era bem isso que estava querendo".

Todo produto de software representa de alguma forma valor para alguém e esse é o principal motivo da existência do software, mas apesar de parecer algo óbvio, muitas vezes o perdemos de vista e acabamos construindo um produto que pode até ser tecnicamente fantástico, mas que não agrega o valor de que o cliente precisa.

É por isso que métodos têm foco em valor de negócio, e o primeiro estágio da fluência ágil diz respeito a agregar valor de negócio ao cliente e a melhorar a visibilidade do trabalho da equipe [57].

O objetivo é aprender a deixar de planejar apenas em termos técnicos, e preocupar-se em planejar levando em conta o valor de negócio que o software desenvolvido poderá agregar ao cliente, assim como refletir e decidir sobre quais as

melhores maneiras de potencializá-lo.

Para dar suporte a esse objetivo as equipes ágeis fazem uso de ferramentas como backlogs, iterações e quadros kanban.

Os principais benefícios alcançados neste estágio são: a transparência, que permite que o progresso (ou a falta de progresso) do time seja facilmente visualizado por todos, inclusive pela gestão, a melhoria contínua para se agregar a cada iteração mais e mais valor, e um melhor alinhamento que resultará em uma equipe mais colaborativa, com menos maus entendidos e menos atrasos desnecessários.

É o momento em que a equipe deverá ganhar maturidade para responsabilizar-se pelo resultado do trabalho, não apenas em termos técnicos, mas também em termos do valor de negócio agregado, e mais do que isso, devem responsabilizar-se pelo resultado do time e não apenas por suas contribuições individuais.

É claro que para que a equipe tenha essa visão de valor em termos de negócio é essencial a presença de alguém com um perfil de negócios próximo para ajudá-los a compreender o que de fato é de valor e o que não é.

Quanto mais cedo for possível agregar valor ao cliente, melhor será o resultado. Quando o cliente recebe o software funcionando em um curtos intervalos de tempo, além de ganhar vantagem competitiva, o trabalho em progresso também diminui (WIP). Consequentemente, a quantidade de trabalho parcialmente pronto diminuirá o desperdício, conforme abordado anteriormente.

A Dell Computadores, por exemplo, não acumula estoques e entrega pedidos personalizados em menos de uma semana e, exatamente por não possuir estoques acumulados, pode oferecer a seus clientes novos produtos antes de seus concorrentes, sem ter que se preocupar em queimar estoques de produtos antigos.

Assim como a faz a Dell, uma equipe de software que entrega rapidamente e mantém um baixo nível de trabalho parcialmente pronto pode responder rapidamente a novas necessidades que venham a surgir sem se preocupar em deixar o trabalho que já foi começado para trás.

As organizações que entregam rapidamente estruturam-se de forma que as pessoas saibam como o trabalho deve ser feito sem que alguém precise dizer a elas o que fazer [48]. Essas pessoas devem ser capazes de resolver problemas e fazer adaptações para responder a mudanças. Essas diretrizes são profundamente aplicadas na Toyota.

Entregar rapidamente é complementar ao princípio de tomar decisões no último momento possível. Por exemplo: se o cliente recebe entregas de software toda semana, não precisa decidir sobre o que será feito daqui a nove meses. Entretanto,

se o cliente recebe software somente a cada três meses, então precisará decidir sobre as novas funcionalidades no mínimo três meses antes de poder vê-las em funcionamento.

Nesse capítulo abordaremos como equipes podem aplicar práticas ágeis para garantir que o cliente receba valor de negócio com frequência através da entrega iterativa de software em funcionamento.

3.1 Disseminando a Visão do Projeto

Projetos sempre nascem de ideias de pessoas. Ideias que, geralmente, nascem de tentativas de tornar um determinado processo mais eficiente. Seus donos são chamados de visionários, porque eles possuem a visão do projeto e, entendem a necessidade e conhecem o problema que o software deve resolver. A visão revela para onde o projeto está indo e por que ele está indo para lá.

Todo projeto possui um ou mais visionários, alguém que conseguiu enxergar uma necessidade e teve uma ideia de solução para suprir essa necessidade. Ainda que um projeto possua muitos visionários, é necessário que alguém tome a responsabilidade de promover e comunicar essa visão à equipe de desenvolvimento e a todos os outros interessados no projeto.

Esse papel, geralmente, é de responsabilidade do gerente de projetos, do próprio cliente, ou de alguém que represente o cliente. No método Scrum, por exemplo, a visão do projeto é responsabilidade do *Product Owner*.

É muito importante que o visionário esteja bem próximo à equipe de desenvolvimento para esclarecer as frequentes dúvidas que surgem ao longo do projeto. Por isso, um dos princípios do método XP é ter o **Cliente Presente**. Uma vez que todos forem contagiados com a visão do projeto, as chances de sucesso se tornam muito maiores.

Há três perguntas essenciais que o visionário precisa comunicar à equipe:

1) Que objetivo o projeto deve atingir?

2) Por que esse projeto agregará valor?

3) Como medir se o projeto foi bem sucedido?

Para que a equipe possa desenvolver um software alinhado com a visão do projeto, é necessário que a visão esteja sempre acessível para todos, por isso a importância do visionário estar presente na equipe.

Muitas equipes mantêm uma breve descrição da visão, como por exemplo, a resposta das três perguntas acima, em um wiki ou portal interno, ou até mesmo impresso e anexado ao quadro da equipe. O importante é que esteja ao alcance de todos.

Os visionários devem estar presentes nos momentos de tomada de decisão, nas reuniões de planejamento, nas apresentações de demonstração das iterações ou reuniões de revisão. Se isso não acontecer, provavelmente a equipe não entenderá o propósito do produto que estão desenvolvendo. Poderíamos comparar isso a um médico que faz uma cirurgia, que não sabe de que mal está tentando livrar o paciente.

Equipes de desenvolvimento de software tomam milhares de pequenas decisões todos os dias, e sem uma boa visão geral e contexto do que está sendo construído, é muito difícil tomar bosas decisões.

Como grande parte das equipes ágeis adotam um único representante da visão do produto, de agora em diante, o visionário será referido como *Product Owner* ou PO.

Será que todos estão na mesma página?

Marc Benioff, CEO da Salesforce.com, conta que um certo dia estavam no elevador um testador, um engenheiro, e um desenvolvedor de sua empresa. De repente alguém de outro conjunto entrou no elevador e perguntou a eles "o que exatamente a Salesforce.com faz?". Para a surpresa do CEO, cada um deu uma resposta diferente [5].

Depois deste evento, Benioff tomou uma série de providências para garantir que todos "estivessem na mesma página" e depois de algum tempo investindo em disseminar essa visão, além de todos ganharem mais assertividade e conhecimento do que estavam fazendo, todos os membros da equipe de tornaram "potenciais marqueteiros e vendedores" do produto.

Quando o projeto possui uma visão única e coesa, a tomada de decisão se torna muito mais fácil, e todos os membros da equipe, de posse da visão, podem participar mais ativamente, oferecer sugestões, e realizar um trabalho mais produtivo e de melhor qualidade.

O Discurso do Elevador

Imagine que você entrou no elevador, e encontra, ninguém menos do que aquele investidor que você sempre sonhou que pudesse se interessar em seu produto. Você tem poucos segundos até que o elevador o deixe em seu andar, e precisa aproveitar a oportunidade para falar sobre o que você e sua equipe estão fazendo. O que você dirá a ele?

A ideia por detrás do discurso do elevador é encontrar uma forma de transmitir a essência de uma ideia em um espaço muito curto de tempo [49]. Fazer esse exercício o ajudará a trazer mais clareza e objetidade para sua visão e será útil para pensar no produto sobre o ponto de vista do cliente e de que forma o produto agregará valor a ele.

A Harvard Business School criou um **website** que ajuda a criar o Discurso do Elevador para seu produto, além de fazer uma análise com sugestões para que você possa melhorar seus discurso [54]. De acordo com a HBS seu discurso deve contemplar os seguintes itens:

1) Quem: O que você mais quer que seu ouvinte se lembre a respeito de sua organização ou produto?

2) O quê: De que forma seu produto agregará valor e trará resultados ou impactará o negócio de seu ouvinte?

3) Por quê: Quais são os benefícios exclusivos, os diferenciais que seu produto oferece?

4) Objetivo: O que você espera que o ouvinte faça?

Criar um discurso do elevador para seu produto pode ser uma forma divertida e rápida de desenvolver um mensagem rápida, objetiva e convincente de apresentar seu produto para os membros da equipe e para potenciais clientes.

3.2 Planejamento e Desenvolvimento Iterativo

O planejamento permite entender a expectativa do cliente em relação ao projeto e traçar estratégias para atendê-las. É o momento de entender as ideias e necessidades do cliente e descobrir as funcionalidades que o software precisará oferecer.

No desenvolvimento ágil de software o time inteiro está envolvido em definir requisitos, otimizá-los, implementá-los, testá-los, integrá-los, e entregá-los aos clientes [37]. Além disso, em um ambiente ágil, planejar é uma atividade constante. Dessa forma, é possível responder rapidamente às mudanças e realizar as adaptações que forem necessárias para que o software seja entregue com o máximo grau de qualidade e eficiência.

Nos métodos ágeis os projetos são divididos em iterações curtas que geralmente têm uma ou algumas poucas semanas de duração. A cada iteração é realizada uma reunião de planejamento para definir o que será feito durante a iteração. O seu objetivo é identificar junto ao cliente o que poderá agregar mais valor ao negócio.

O processo de planejamento no método XP é chamado de "Jogo do Planejamento", no Scrum simplesmente de Planejamento. O objetivo do jogo do planejamento é garantir que o máximo de valor possível será gerado no dia a dia de trabalho através da criatividade e das habilidades da equipe.

O planejamento da iteração acontece em uma reunião no início de cada iteração. É importante que nesta reunião estejam presentes a equipe de desenvolvimento e o cliente. Assume-se que o cliente tem toda a informação necessária no que diz respeito ao que agrega valor ao software, e que a equipe de desenvolvimento tem toda a informação necessária em relação a quanto custa para agregar tal valor.

A grande sacada é minimizar os custos e maximizar o valor agregado. Através da estimativa, os desenvolvedores podem dizer quanto custa, em outras palavras, quanto tempo leva para desenvolver determinada funcionalidade; e através da priorização, o cliente pode definir quanto valor uma história pode agregar, ou seja, quais funcionalidades podem trazer mais benefícios. Temos então uma relação de custo/benefício.

Através dessa colaboração entre a equipe de desenvolvimento e o cliente é possível traçar um plano para atingir sucesso no projeto. O jogo do planejamento é dividido em duas fases: o planejamento de *releases* e o planejamento de iterações.

Um dos principais benefícios do planejamento iterativo é a eliminação do desperdício de uma longa análise prematura, que é considerado um dos maiores desperdícios no desenvolvimento de software [49]. É comum em projetos waterfall que a análise de todos os requisitos seja feita de antemão, de forma que no momento do desenvolvimento os requisitos já mudaram e da maneira que foram analisados já não podem mais agregar valor ao cliente. Essa prática de levantamento prematuro de requisitos é conhecida como *Big Requirements Up Front* (BRUP).

Backlogs DEEP

Para ajudar *Product Owners* a não caírem no erro do levantamento prematuro de requisitos, Mike Cohn, criou o acrônimo DEEP [20] que representa 4 qualidades de um bom *backlog*:

1) D - Detalhado Apropriadamente: Os itens de maior prioridade devem ser mais detalhados e os itens de menor prioridade não precisam de muitos detalhes.

2) E - Estimado: Se todos os itens do Backlog estiverem estimados pela equipe, o PO terá mais facilidade em priorizar o *backlog* e terá melhor previsibilidade do projeto.

3) E - Emergente: O *Product Backlog* é um artefato vivo, isto é, deve mudar frequentemente de acordo com as mudanças de negócio. Novos itens são incluídos, itens antigos são removidos ou modificados.

4) P - Priorizado: Os itens do *backlog* devem estar priorizados do maior valor de negócio para o menor valor de negócio para que os itens mais importantes sejam detalhados e implementados antes dos menos importantes.

Além disso, em abordagens tradicionais, era muito comum que os requisitos fossem transmitidos ao desenvolver apenas por escrito através de um documento. Em contrapartida, a essência da comunicação nos métodos ágeis é a **comunicação face a face**.

De acordo com Alistair Cockburn, a comunicação através de papel é fria e pobre, porque não há oportunidade para perguntas e respostas, a comunicação entre pessoas face a face é quente e muito mais rica porque oferece a oportunidade de iteração entre as partes [17].

É claro que isso não torna, de forma alguma, a comunicação escrita irrelevante, pelo contrário, ela também é importante, mas não é suficiente, e nem mais eficiente em todos os cenários.

É por isso que muitos métodos ágeis defendem a ideia de uma reunião de planejamento em que a equipe e o *Product Owner* participem juntos, e que, através de

uma conversa, criem histórias de usuário que representam as funcionalidades a serem implementadas no software.

Ao decorrer deste capítulo, o conceito de histórias de usuário será mais amplamente explorado, mas, antes disso, explicaremos um pouco mais o que se faz em um planejamento de iteração.

3.3 Planejando uma Iteração

O planejamento de iterações permite estruturar as atividades do dia a dia da equipe. Um *release* pode conter várias iterações (figura 3.1), de forma que as histórias que precisam ser entregues no *release* são incluídas em uma ou mais iterações. Durante a iteração as histórias são desenvolvidas e no final se tem um software desenvolvido e testado e, ao terminar uma iteração, uma nova iteração é iniciada.
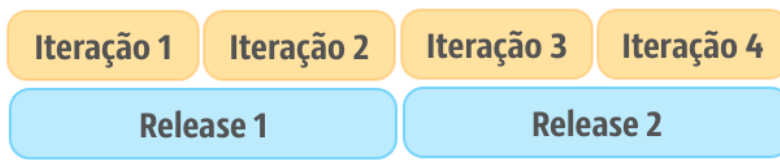
Figura 3.1: Releases e Iterações

As iterações geralmente têm periodicidade semanal, porém, tanto o tempo das iterações como dos *releases* podem ser definidos de acordo com a necessidade de cada organização. O tamanho ideal para uma iteração é o menor tempo possível que seja necessário para que a equipe consiga entregar algo de valor com qualidade para o cliente.

Ao planejar uma nova iteração, é importante ter em mãos o resultado da iteração passada, isto é, a soma das estimativas das histórias que foram desenvolvidas. Esse é o número de pontos de estimativa que se pode esperar para a próxima iteração. Essa soma de pontos é conhecida como velocidade da equipe. Há quem utilize uma média das velocidades das iterações passadas para definir quantos pontos de complexidade serão inseridos na nova iteração e há quem prefira utilizar apenas a velocidade da última iteração.

Para inserir as histórias na iteração é preciso que elas estejam estimadas em complexidade e priorizadas segundo o valor de negócio, o que provavelmente já terá sido

feito no planejamento de *release*. Dessa forma, selecionar as histórias para a iteração, geralmente, é algo rápido. O cliente verifica história por história, e as inclui na nova iteração de acordo com a prioridade. A somatória de pontos das histórias selecionadas, idealmente, não deve ser maior do que a definida, seja com base na iteração passada ou na média das anteriores.

Depois de selecionadas quais histórias serão implementadas na nova iteração, é preciso explicá-las à equipe. Nesse momento é importante que o PO ou pessoas de negócio possa estar presentes para que possam esclarecer aos desenvolvedores suas dúvidas de negócio, permitindo que ao final da reunião os desenvolvedores estejam preparados para implementar estas histórias.

Se a equipe conseguir entregar todas as histórias antes do fim da iteração, é preciso solicitar ao cliente mais histórias. Então, o cliente escolhe quais deverão ser adicionadas à iteração considerando o tempo restante e sua prioridade. Na próxima iteração o cliente poderá incluir um número maior de pontos, considerando que a velocidade da equipe melhorou. Isso geralmente acontece à medida que a equipe vai amadurecendo e ganhando mais domínio sobre a tecnologia utilizada no projeto e nos conceitos de negócio.

Um problema muito comum que pode afetar a ordem de desenvolvimento das histórias são as dependências técnicas entre elas. É possível que o cliente determine que uma história específica tenha prioridade em relação à outra, porém, tecnicamente pode ser mais viável fazer o contrário. Nesses casos é importante que a equipe seja sincera com o cliente em relação às dificuldades técnicas e apresente a ele os benefícios de implementar as histórias na ordem inversa. Entretanto, é sempre recomendado que a equipe permita que o cliente escolha o que ele quer que seja entregue primeiro e se esforce ao máximo para entregar de acordo com a prioridade determinada por ele.

Em se falando de XP e Scrum, não é recomendado adicionar novas histórias a uma iteração que já tenha sido iniciada. É comum que ao longo de uma iteração o cliente solicite que alguma nova história seja adicionada com urgência. Atender a este tipo de solicitação emergente e não planejada pode impactar negativamente no andamento do trabalho da equipe e, como mencionado anteriormente, pela mesma razão, não é recomendado alterar as histórias de uma iteração em andamento. Se houver muitos casos de mudanças de escopo dentro de uma iteração, e isso, realmente, for importante para o negócio, talvez seja o caso de avaliar utilizar um método ágil de fluxo contínuo ao invés de iterativo, como Kanban, por exemplo.

Alterar uma iteração em andamento pode significar trabalho perdido, desgaste,

perda da concentração e atrasos. No entanto, responder rapidamente às necessidades de negócio do cliente é um valor ágil e algo que deve ser sempre levado em consideração.

Em casos como este, é possível negociar com o cliente, e depois de conscientizá-lo da problemática de alterar iterações em andamento, incluir a história emergente com a condição de retirar da iteração outra história com um valor de estimativa de esforço igual ou próximo. Deve-se evitar, é claro, retirar histórias que já tenham seu desenvolvimento iniciado.

O cliente, provavelmente, ficará contente por sua solicitação emergente ter sido atendida e mais valor ter sido agregado ao seu negócio. Porém, é preciso ter cuidado para que essas emergências não se tornem algo constante e prejudicial. Solicitações emergentes devem ser esporádicas. É preciso insistir para que o cliente tenha disciplina e respeite as iterações para que as coisas possam caminhar conforme o planejamento.

Outros tipos de demandas não planejadas podem surgir ao longo da iteração. Um exemplo muito comum são os indesejáveis defeitos. Por maiores que sejam as atitudes preventivas da equipe para manter a qualidade do software com testes e integração contínua, é comum que defeitos se apresentem aqui ou ali. Nesses casos pode ser necessário que a equipe pare o trabalho e tome providências para sanar o problema.

A lógica é simples: se uma determinada funcionalidade já está em produção, significa que ela era mais importante do que as outras que ficaram pendentes, por isso foi priorizada e desenvolvida antes das outras. Caso um defeito surja e e essa pare de funcionar, provavelmente, será mais importante corrigir o defeito, do que entregar algo que tinha menos prioridade.

Vale a pena acompanhar as demandas de trabalho não planejado que surgem durante uma iteração. Se for muito frequentes fique atento: algo provavelmente está errado ou talvez seja mais interessante avaliar trabalhar com fluxo contínuo em vez de iterações.

3.4 A Reunião Diária

Estabelecer uma comunicação eficiente entre os membros da equipe é extremamente importante. As reuniões diárias, conhecidas no XP como "*Stand Up Mettings*" (Reuniões em Pé) e no Scrum como *Daily Scrum* (Scrum Diário) são práticas muito eficazes para que todos estejam sempre a par do *status* atual do projeto.

Todos os dias em um horário preestabelecido, geralmente no início do dia, a equipe se reúne por mais ou menos 15 minutos para sincronizar o estado do projeto e manter todos informados sobre os últimos acontecimentos e como as coisas estão ou não evoluindo.

Através da reunião diária todos os membros da equipe ficam cientes do andamento do projeto e podem aproveitar a oportunidade para apresentar suas dificuldades e pedir ajuda aos outros membros.

No Scrum, por exemplo, cada membro do time responde a três perguntas na reunião diária:

1) O que eu fiz desde a última reunião (ontem)?

2) O que farei até a próxima reunião (até amanhã)?

3) Que problemas estão me impedindo de progredir?

Tim Ottinger, sugere outras 4 perguntas que têm um foco maior em entrega, aprendizagem e colaboração [44]:

- Que histórias você ajudou a terminar desde a última reunião diária?
- Que história você ajudará a terminar hoje?
- Como o resto do time pode a ajudar a empurrar uma história para "pronto".
- O que você aprendeu de novo?

Um fator importante que deve ser considerado para uma boa comunicação é o número de integrantes da equipe: quanto mais pessoas, mais difícil se torna a comunicação. Por isso, equipes ágeis geralmente são compostas por um número de membros que varia de cinco a nove pessoas. No capítulo 6 abordaremos com mais profundidade algumas boas práticas para formação de equipes.

É importante que todos tenham oportunidade de falar na reunião diária, nos primeiros dias é importante que alguém atue como facilitador (geralmente alguém com papel de scrum master) para assegurar que ninguém domine a reunião e fale sozinho durante os 15 minutos, ou que alguém deixe de falar.

Para criar essa disciplina, algumas equipes utilizam alarmes que tocam de 2 em 2 minutos, por exemplo, para evitar que alguém fale demais. Outras passam um bola de mão em mão até que todos a bola volte a pessoa que falou primeiro na reunião. Ao

passo que a equipe compreende os benefícios de uma boa reunião diária, esse tipo de ferramenta, geralmente, torna-se desnecessária, mas pode ser útil para começar.

Ken Schwaber, ressalta que é importante que a reunião diária seja realizada em um local conhecido por todos [55], e que se deve evitar entrar em discussões que fujam do contexto das três perguntas principais como, por exemplo, discutir sobre problemas, abordagens de design, as notícias do jornal do dia etc. Se um problema que requer discussão for identificado, se necessário, deve-se marcar com os interessados uma reunião para se discutir após a reunião diária ou em outro momento.

Porcos e Galinhas
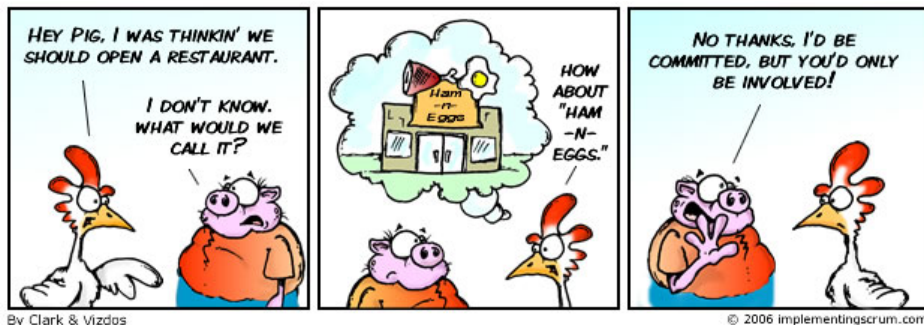
Figura 3.2: Porcos e Galinhas

A galinha diz para o Porco:

- Ei Porco, eu estava pensando... Nós deveríamos abrir um restaurante juntos.

O porco responde:

- Não sei não... Qual seria o nome?

A galinha responde:

- Que tal "Ovos com Bacon"?

O porco responde:

- Não obrigado. Eu estaria comprometido, e você estaria apenas envolvida.

No Scrum todos as atividades de gestão do projeto são dividas entre os papéis de Scrum Master, *Product Owner* e Time, essas são as pessoas comprometidas como o projeto. O Scrum defende que aqueles que estão comprometidos devem ter autonomia para fazer o que for preciso para o sucesso do projeto, e que aqueles que estão apenas envolvidos não façam intervenções desnecessárias.

Na reunião diária, por exemplo, no Scrum, as galinhas não podem falar, fazer observações, fazer caretas ou intervir, se quiserem participar devem ficar na periferia apenas com ouvintes. Isso permite que o time fique focado no que é importante sem explicações ou debates desnecessários.

3.5 Limitando o Trabalho em Progresso

"Pare de começar e comece a Terminar"
– Sterling Mortensen

Discutimos anteriormente que estoque em desenvolvimento de software, é representado por trabalho que foi começado, porém ainda não foi terminado e, de acordo com o pensamento lean, isso é desperdício, e deve ser eliminado.

Existe uma relação entre o trabalho em progresso (WIP) e o *lead time*. É uma relação linear proporcional, isso quer dizer que o *lead time* cresce à medida que o trabalho em progresso cresce, e diminui a media que o trabalho em progresso diminui. Essa relação é conhecida na industria como "a lei de little" [50].

Definimos o limite do trabalho em progresso no quadro adicionando limites explícitos que definem quantos itens podem estar em progresso em cada estado do fluxo de trabalho, ou seja, cada coluna do quadro. Veja na figura 3.3:

Figura 3.3: Quadro com WIP

Reduzir o trabalho em progresso, reduz também o **lead time**, permitindo que se entregue com mais frequência. Para isso é importante definir se será ou não permitido que cada membro trabalhe em mais de uma tarefa ao mesmo tempo.

Trabalhar paralelamente em duas tarefas pode levar a problemas como interrupções, falta de foco, desperdício de tempo na mudança de contexto, ou até mesmo na mudança de ambiente de desenvolvimento, coisas que levam à perda de produtividade. Experimentos empíricos devem ser feitos em cada equipe para verificar o que funciona melhor.

Os limites podem ser definidos inicialmente com um pouco mais do que a quantidade de pessoas que trabalha naquela etapa do processo. Numa equipe com dois testadores, por exemplo, poder-se-ia iniciar com um limite de 3, e fazer experimentos ao longo do tempo para se chegar no limite ideal.

No quadro da figura 3.3, o limite do desenvolvimento está estourado, e os desenvolvedores não podem puxar mais histórias, nem os testadores que estão no limite. Nessa situação a equipe toda precisa focar em entregar as histórias que estão mais próximas do final do processo antes de poder começar a trabalhar em novas histórias.

Sem limite de trabalho em progresso, mais trabalho seria começado, e poderia-se correr o risco de na data de *release* a equipe não ter terminado ainda uma série de histórias e defeitos que mesmo estando em andamento não estariam prontos para serem entregues ao cliente.

Quando uma história ficar pronta no estágio em que se encontra e, esperando para ser puxada para o próximo estágio, está em uma fila. As filas devem ser tão pequenas quanto possível. Recomenda-se considerar a criação de pequenas filas para que absorvam a variação do processo e permitam que fluxo não seja interrompido.

Muitos processos, se não possuírem filas, poderão fazer com que as pessoas fiquem desocupadas, gerando tempo ocioso, devido a variações no tempo que cada item leva para ficar pronto. Novamente, é importante observar seu processo e, empiricamente, optar pela solução que permitir melhor fluidez.

3.6 Escrevendo Histórias de Usuário

Histórias de usuário são descrições simples e curtas de funcionalidades que deverão ser implementadas no software. As Histórias são, geralmente, escritas pelo próprio *Product Owner*, com suas próprias palavras e, então, são registradas em cartões que posteriormente são anexadas ao quadro da equipe.

As histórias contêm apenas uma breve descrição que representa uma necessidade do cliente, porém, apesar da simplicidade da história, o cliente deve possuir o conhecimento necessário para disponibilizar informações de negócio para que a equipe possa transformar sua necessidade em software. Por exemplo, imagine uma história cuja funcionalidade seja desenvolver um relatório de balanço contábil; será necessário que o cliente possua o conhecimento de negócio necessário, isso é, saiba o que é um balanço contábil e quais são suas características, para que assim possa explicar à equipe o que exatamente devem desenvolver.

Toda história de usuário deve possuir um critério de aceitação, também definido pelo cliente, que permite à equipe identificar quando ela está implementada por completo e com sucesso. Ao escrever uma história o cliente assume responsabilidade sobre ela.

As histórias de usuário têm caráter de negócio, por isso, "instalar o servidor de integração contínua" ou "atualizar a versão da biblioteca de injeção de dependências", geralmente, não são exemplos válidos. Podem até ser exemplos de tarefas, mas não de histórias de usuários. Veremos a diferença entre histórias e tarefas mais adiante.

Alguns exemplos de história de usuários válidas em um projeto de desenvolvimento de um site de relacionamentos podem ser:

- Um usuário pode publicar suas fotos em seu perfil;
- Usuários podem criar e participar de comunidades;
- Comunidades devem possuir moderadores.

Mike Cohn propôs um formato interessante que é amplamente conhecido e utilizado no mercado:

"Eu, como um (papel) quero (funcionalidade) para que (valor de negócio)."

Se aplicarmos o modelo proposto nas histórias citadas anteriormente, elas ficariam da seguinte forma:

- Eu como um forista quero publicar minha foto em meu perfil para que os outros participantes possam me identificar mais facilmente.
- Eu como um usuário gostaria de criar uma comunidade para que eu possa reunir pessoas com interesses semelhantes aos meus.
- Eu como um moderador gostaria aprovar ou rejeitar respostas a tópicos do forum para evitar postagens inadequadas.

Investindo em Boas Histórias

Segundo Mike Cohn, autor do livro: "User Stories Applied", boas histórias devem conter as seis características seguintes:

1. Histórias devem ser independentes para evitar problemas de planejamento e estimativa. Se duas histórias possuírem dependência entre elas, e uma tiver prioridade alta e a outra baixa, provavelmente a história dependente de prioridade baixa

precisará ser desenvolvida antes das outras tarefas, assim que chegar a hora da história de prioridade alta ser desenvolvida. Nesses casos, pode ser interessante unir ambas em uma única história;

2. Histórias devem ser negociáveis, não devem ser vistas como requisitos que deverão ser implementados a qualquer preço. Lembre-se, histórias são apenas curtas descrições de funcionalidades que deverão ser analisadas e discutidas com o cliente no momento em que sua prioridade for atingida, ou seja, no momento em que a história fizer parte de uma iteração. É importante que a história seja detalhada apenas quando o momento de seu desenvolvimento estiver próximo, isto é, quando chegar o momento de a história entrar na iteração corrente. Quanto mais cedo ela for analisada e detalhada, maiores serão as chances de haver retrabalho e até mesmo perda de tempo e energia, considerando que o cliente aprende e muda de ideia à medida que as iterações vão sendo concluídas.

Para ilustrar melhor este exemplo, imagine uma história que tenha como objetivo a criação de um relatório de apuração de impostos. Imagine que, segundo sua prioridade, ela será implementada somente daqui a seis meses, mas que ainda assim, inicia-se um trabalho de análise e discussão sobre ela. Se o governo realizar alguma mudança no plano tributário, você provavelmente terá desperdiçado tempo e dinheiro!

3. Histórias devem agregar valor aos clientes. Como mencionado anteriormente, histórias devem descrever funcionalidades que de alguma forma oferecerão ao cliente retorno ao seu investimento. Por essa razão, histórias de caráter técnico não são aconselháveis, por exemplo, "atualizar o Spring Framework para versão 2.5" não agrega valor de negócio ao software, pelo menos não diretamente, isso não significa que você não possa fazê-lo, mas é aconselhável que exista uma história escrita pelo cliente que a justifique;

4. Histórias devem ser estimáveis. É importante que os desenvolvedores sejam capazes de estimá-las. Para que sejam estimáveis é necessário que os desenvolvedores possuam ou tenham acesso através do cliente ao conhecimento de negócio, e possuam também o conhecimento técnico necessário para transformar a história em software. Seria inviável pedir a um desenvolvedor que nunca escreveu uma linha de código em Java que estime o esforço para desenvolver um pedido de venda em um projeto Java EE com EJB 3. O mesmo aconteceria se pedíssemos ao desenvolvedor para que estimasse o esforço para criar um relatório de fluxo de caixa, sem antes explicar a ele o que é um fluxo de caixa;

5. Histórias devem ser pequenas. Histórias muito grandes são difíceis de esti-

mar, por isso deve-se tomar os devidos cuidados para mantê-las sempre curtas e, quando necessário, quebrá-las em histórias menores. Para ilustrar melhor, imagine a seguinte história: "Um usuário pode realizar compras na loja virtual". Ela pode envolver muitas funcionalidades que estão implícitas, e por isso, pode gerar diferentes interpretações e levar os desenvolvedores ao erro. Em casos como este, podemos quebrar a história grande em outras histórias menores: "Um usuário pode adicionar produtos ao seu carrinho de compras", "Ao finalizar a compra o usuário deve escolher a forma de pagamento", "Para compras acima de R$100,00 o frete deverá ser gratuito" etc.;

6. Histórias devem ser testáveis. É preciso ter critérios de aceitação bem definidos para que um desenvolvedor possa saber quando uma história está ou não concluída. Uma boa prática ao usar cartões para escrever as histórias é pedir ao cliente que escreva alguns critérios de aceitação no verso do cartão.

Para que a história seja testável, é preciso que o cliente escreva critérios objetivos e concretos. Por exemplo, "O relatório deve ser intuitivo" é um exemplo de critério difícil testar, porque o conceito de intuitivo é abstrato e subjetivo, pode possuir significados diferentes para cada pessoa. Ao invés disso, "o relatório deve possuir um rodapé com a soma dos valores" ou "o título das colunas deve se manter fixo ao navegar entre os lançamentos" são exemplos de critérios fáceis de testar, bastando que o desenvolvedor certifique-se de que eles estão sendo atendidos para saber se a história será aceita pelo cliente.

Essas características formam o acrônimo INVEST:

- I - Independente
- N - Negociável
- V - Valiosa
- E - Estimável
- S - Sucinta
- T - Testável

Temas, Épicos, Funcionalidades e Histórias

É comum que muitos projetos de software, naturalmente, possuam uma estrutura hierárquica de requisitos, de forma que algumas necessidades do usuário de alto

nível podem ser derivadas em diversas partes menores que, por sua vez, podem ser dividas novamente até uma granularidade que possa ser mais facilmente compreendida e que possa ser desenvolvida em um determinado espaço de tempo.

Dean Leffingwell, sugere que a hierarquia de requisitos comece com **Temas**que representam um conjunto de iniciativas que guiam os investimentos em sistemas, produtos, serviços e aplicações [37]. Desses temas são derivados os épicos, iniciativas de desenvolvimento que têm potencial de agregar valor ao tema do qual foi derivado. São requisitos em alto nível e não são implementados diretamente, antes, são quebrados em funcionalidades, que posteriormente são quebradas em histórias de usuário (veja na figura 3.4).
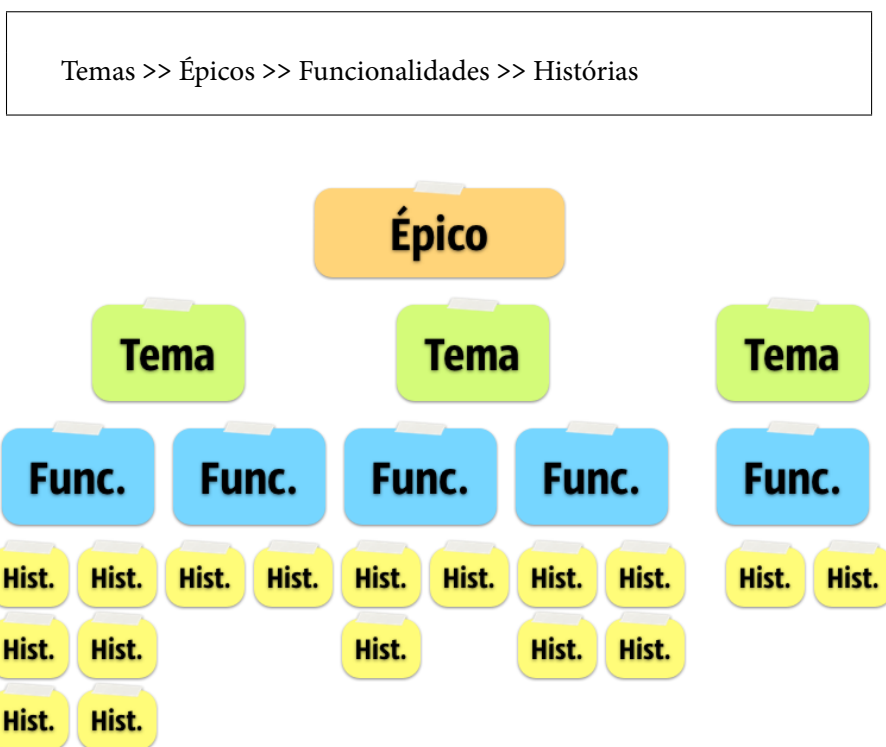
Figura 3.4: Hierarquia de Requisitos

Épicos são requisitos grandes que, geralmente, não podem ser implementados em uma única iteração e, normalmente, contemplam diversos eventos, fluxos e cenários. Por questões de organização e priorização, muita vezes, é interessante manter

esses itens no **backlog**, sem detalhar exatamente o que deverá ser feito, e então, somente quando a prioridade do épico for aumentando e a hora de trazê-lo para o planejamento estiver se aproximando, será o momento certo para dividi-lo em histórias de usuário.

Para melhor ilustrar imagine um sistema de gestão empresarial. O sistema pode ter temas como Comercial, Financeiro e Contábil, por exemplo.

O tema Financeiro pode ser derivado em Épicos como Cobranças e Pagamentos, já o Épico Pagamentos pode ser derivado em funcionalidades como Pagamento por Boletos e Pagamentos por Depósitos.

A Funcionalidade Pagamento por Boleto, poderia ser derivada em diversas histórias, como, por exemplo, Pagamento de Boletos Vencidos, Pagamento de Boletos com Juros etc.

Dividindo Histórias

É comum que algumas histórias de usuários sejam grandes demais para serem implementas em uma única iteração. Nesses casos, é interessante dividi-las em várias histórias menores. É importante, porém, manter o aspecto INVEST. Você pode dividir histórias por variações nas regras de negócio, por variações nos dados, por diferentes cenários de uso etc.

Quebrando Histórias em Tarefas

De um modo geral, histórias requerem certo esforço para serem implementadas. Por isso elas são dividas em tarefas, que representam os passos necessários para que a funcionalidade da história seja desenvolvida e entregue em funcionamento ao cliente. É recomendado que as histórias sejam quebradas em tarefas que não precisem de mais de um dia de trabalho para serem desenvolvidas. Posteriormente veremos como isso acontece e como as atividades técnicas são gerenciadas.

Apenas depois de ouvir o PO e compreender a história, a equipe a quebra em partes menores, chamadas simplesmente de tarefas. Neste momento a presença do PO é menos importante, porque as tarefas têm um âmbito mais técnico, e geralmente terão descrições como: criar as tabelas no banco de dados, criar objetos de negócio, realizar integração através de web services etc.

Não é preciso entrar em detalhes minuciosos sobre cada tarefa e neste momento pode-se tomar algumas decisões de **design** `que a equipe considerar importante, como linguagens,` **frameworks**, bibliotecas e padrões a serem

utilizados, além de questões que envolvam performance e escalabilidade. Decisões que a equipe considerar menos importantes podem ser tomadas mais tarde, apenas no momento da implementação.

3.7 Mapeando histórias de usuários

Mapeamento de histórias é uma técnica que consiste basicamente em decompor atividades de usuário de alto nível em um **workflow** que pode ser decomposto posteriormente em um conjunto de histórias de usuário. A técnica, que foi popularizada por Jeff Patton, é centrada na perspectiva do usuário [53].

Primeiramente, temos os épicos representando requisitos de alto nível, depois temos as funcionalidades, e finalmente as histórias 3.5:
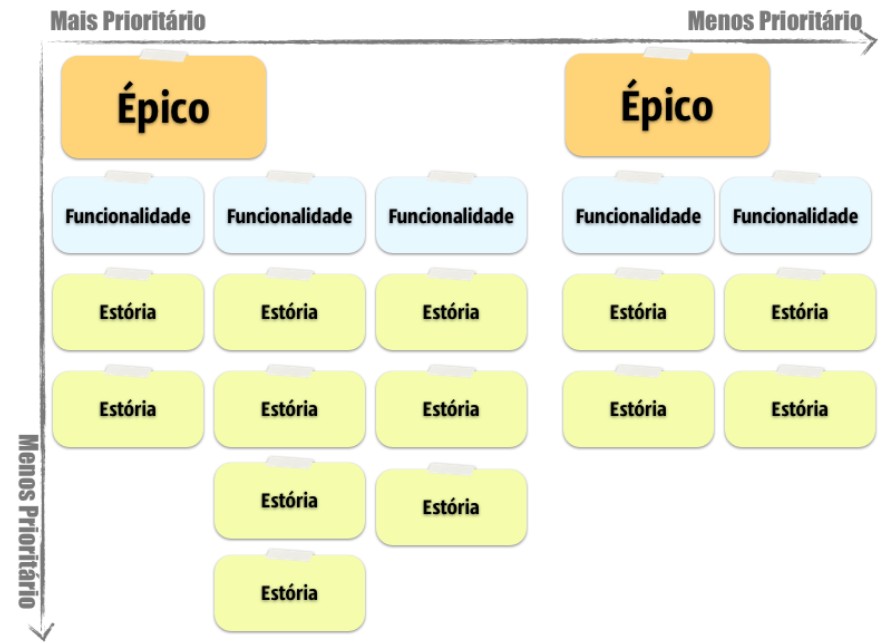
Figura 3.5: Mapa de histórias

Dependendo da abordagem os nomes dos níveis de requisitos podem mudar. Na abordagem original de mapeamentos de histórias, Patton usa Atividades, Tarefas e Subtarefas em vez de épicos e funcionalidades e histórias. Outras fontes entendem

que épicos são maiores que temas, por exemplo, mas isso não importa muito: uma vez compreendida a essência da prática, você pode escolher a semântica que for mais adequada ao seu contexto.

Note que tudo está priorizado em dois eixos, tanto da esquerda para a direita, que representa a prioridade entre diferentes épicos, e diferentes funcionalidades, quanto de baixo para cima que representa a prioridade entre diferentes histórias de usuário.

Depois de pronto e priorizado, você pode utilizar o mapa para definir *releases* em camadas, ou seja *releases* que contêm um pouco de cada de funcionalidade (veja na figura 3.6.

Figura 3.6: Releases no Mapa de Histórias

Para ilustrar, imagine, por exemplo, um site de comércio eletrônico que será composto por funcionalidades de carrinho de compra, vitrine eletrônica e Consulta de Pedidos. Em vez de entregar uma funcionade completa com todas as histórias envolvidas, é possível entregar algumas histórias de cada funcionalidade por *release*.

Dessa forma, o produto poderia ser lançado com o básico de cada funcionalidade e já poderia ser utilizado. À medida que os outros *releases* forem entrando no ar, as funcionalidades seriam aprimoradas com a entrega de outras histórias.

Recomendações

- Desenvolva o mapa de histórias colaborativamente, convide o PO, analistas de negócio, desenvolvedores etc.
- Mantenha o mapa sempre visível para que o time possa discutir frente a ele, e tomar decisões mais facilmente uma vez que o mapa dá uma visão melhor do "todo" e sobre como as diferentes funcionalidades e histórias de usuário estão conectadas a objetivos de negócio [43].

3.8 Conhecendo os Usuários através de Personas

Personas são perfis baseados nos clientes e usuários do produto. A criação de personas ajuda a identificar Perfis de Clientes e Usuários importantes para o sucesso do produto e a melhorar o entendimento, a comunicação e a tomada de decisão do time sobre os diferentes usuários, de forma a otimizar suas funcionalidades para o perfil do usuário final, e garantir que as necessidades de diferentes perfis sejam bem atendidas pelo produto em desenvolvimento.

Veja uma exemplo de persona na figura 3.7:

Figura 3.7: Persona

Modelo para Criação de Personas

Não há uma forma certa ou errada de se desenvolver personas, mas essas são algumas características comuns que podem ajudar o time a compreender um perfil de usuário. Você pode escolher algumas dessas para começar a desenvolver as personas de seu produto:

- Nome
- Papel
- Valores
- Objetivos
- Problemas
- Necessidades
- Desejos
- Expectativas
- Preferências
- Padrões de Comportamento
- Casos de Uso
- Atividades

Depois de criadas as personas do produto, é interessante associá-las às histórias de usuário, para que o time saiba quem é o usuário interessado na história a ser desenvolvida. É possível fazer isso tornando a persona explicita na descrição da história:

Como um **contador** eu gostaria de buscar os últimos lançamentos...

Como um **gerentes de contas** eu gostaria de atualizar o saldo...

Como um **consumidor** eu gostaria de...

3.9 Melhorando a Previsibilidade com Estimativas

A estimativa permite que a equipe meça a produtividade e saiba quanto tempo será necessário para concluir o desenvolvimento das histórias do projeto, além de ajudar o cliente a entender qual o custo do desenvolvimento de cada história e qual é a média de tempo que será necessário para que a funcionalidade seja desenvolvida e entregue.

Estimativas são complexas e não são precisas. Uma série de fatores devem ser levados em consideração, entre eles, a maturidade da equipe em relação ao projeto e às tecnologias aplicadas nele.

É comum que no início do projeto funcionalidades de complexidades semelhantes levam mais tempo para serem implementadas do que no final do projeto. Isso ocorre porque a equipe ganha produtividade à medida que aprende mais sobre o negócio e sobre a tecnologia utilizada. Assumindo essa premissa, é possível concluir que a estimava de uma determinada história pode mudar ao longo do tempo.

É importante que todos os membros da equipe participem do processo de estimar. Quando um desenvolver estima uma história sozinho, ele o faz com base em suas experiências pessoais. Quando a estimativa é realizada em equipe, o **expertise** envolvido é muito maior e as chances de obter um resultado mais adequado também.

Uma das medidas mais comuns de estimativa são horas. A equipe indica quantas horas serão consumidas para que uma funcionalidade seja implementada. No desenvolvimento ágil, é mais comum utilizar pontos para realizar estimativas.

Pontos podem ter significados diferentes de acordo com cada equipe ou projeto. Uma das formas, como recomendado no método XP, é quantificar um ponto como sendo um dia ideal de trabalho, ou seja, um dia totalmente dedicado a realizar uma tarefa, sem interrupções. Dessa forma, dois pontos seriam dois dias e assim por diante.

Outra forma, recomendada pelo método Scrum, é utilizar pontos relacionados ao esforço, em vez de tempo. Dessa forma, toma-se uma funcionalidade simples como referência, por exemplo, o desenvolvimento de uma tela de cadastro, e assume-se que essa referência vale dois pontos. Ao estimar outra tarefa avalia-se em quantas vezes mais fácil ou mais difícil em relação à referência é a história atual.

Outra técnica para tentar absorver a incerteza e falta de precisão das estimativas é a utilização de intervalos, como a sequência de Fibonacci: "0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144" ou uma sequência simplificada "0, 1, 2, 3, 5, 8, 13, 20, 40, 100". Assim, utiliza-se apenas os valores dentro da sequência para representar o esforço necessário para a implementação da história.

No Scrum, utiliza-se o Planning Poker, ou Poker de Planejamento. Nesta técnica cada membro da equipe recebe um conjunto de cartas com os valores de uma sequência, como as descritas anteriormente. Em seguida todas as histórias são analisadas separadamente, e cada membro da equipe coloca uma carta sobre a mesa com o número de pontos que considera que a história consumirá.

Quando houver grandes diferenças, as pessoas que jogaram as cartas de maior e menor valor explicam suas razões para escolher aquele valor e, então, com base nas explicações, as cartas são jogadas novamente até que um consenso seja encontrado e a estimativa seja definida.

Na realidade, não importa qual técnica sua equipe utiliza, o importante é que todos os membros da equipe participem e que seja definido um valor de esforço ou custo para que seja possível medir a velocidade da equipe para fundamentar o planejamento de *releases* e dar uma certa previsibilidade para os interessados (**stakeholders**).

3.10 Definindo Entregas com o Planejamento de Releases

O objetivo das reuniões de planejamento de *releases* é definir quando novos incrementos do projeto serão, finalmente, instalados em produção. Assim como os outros procedimentos, esta reunião também deve possuir uma cadência; geralmente, a cada duas semanas ou mensal.

Todos os interessados do projeto podem participar da reunião. Especialistas de integração, especialistas de redes, desenvolvedores, testadores, designers e todos que estiverem relacionados ao processo de entrega (**deployment**) devem estar presentes. O importante é que a reunião seja composta pelas pessoas capazes de tomar decisões técnicas, de negócio e gerenciais.

Quando o processo de *release* é muito complexo, pode ser interessante ter uma pessoa responsável por coordená-lo. É comum que um gerente de projetos assuma esse papel, liderando a reunião, além de coordenar os *releases*.

Durante a reunião de planejamento de *releases* deve-se deliberar sobre que itens irão compor o próximo *release*. Este é um momento oportuno para se levantar e considerar os ricos, pensar sobre planos de contingência e treinamentos, analisar quanto tempo o processo de deploy deverá levar, quem deverá estar presente no momento do deploy e quando ele será realizado.

Para que os *releases* sejam curtos é necessário tentar dividir as histórias das iterações em grupos que representem o mínimo possível de funcionalidade que agregue algum valor. Dessa forma o cliente não precisará esperar muito tempo para receber retorno de seu investimento, e receberá esse retorno frequentemente em funcionalidades que seriam adicionadas pouco a pouco ao produto, agregando assim cada vez mais valor ao software.

O planejamento de *releases* oferece um mapa para alcançar o objetivo do projeto. Imagine um projeto de seis meses para um supermercado. Poderíamos dividi-lo em seis *releases* de um mês cada. No primeiro *release* poderíamos entregar o módulo de gerenciamento de preços, no segundo *release* o módulo de pedido de compra, no terceiro os relatórios gerenciais, e assim por diante. Assim, a cada mês o cliente poderia resgatar um pouco de seu investimento em forma de software.

É importante buscar entregar o máximo de valor agregado ao cliente a cada *release*, mas isso não significa o mesmo que entregar o maior número de funcionalidades. É preciso entender o que agregará mais valor.

Muitas vezes uma única funcionalidade pode agregar mais valor do que dezenas de outras juntas. No Scrum, por exemplo, fala-se do conceito de valor de negócio ou *business value*, que pode ser representado por um intervalo numérico, como por exemplo de 0 a 100. O valor de negócio é definido pelo PO, e é através dele que a prioridade das funcionalidades a serem desenvolvidas é determinada. Esse valor pode mudar ao longo do tempo, por isso é importante estar preparado para adaptar o planejamento.

Outro fator importante em se tratando de priorização é detalhar e quebrar apenas os requisitos de maior importância para que seja mais simples se manter as histórias priorizadas e organizadas. Por exemplo, se você está desenvolvendo um sistema financeiro, e sabe que não fará o calculo de financiamento nos próximos 5 *releases*, é melhor manter apenas um épico "Financiamento" do que já quebrar em histórias específicas "Calcular Financiamento com Sistema SAC", "Calcular Financiamento com Sistema Price" etc.

Mantenha o Backlog Pequeno

Para se ter uma ideia de como a complexidade de priorizar histórias aumenta à medida que se adiciona histórias no backlog: se você tiver 3 estórias no backlog (A, B, C), você pode ter 6 ordens de prioridade diferentes (A, B, C), (B, A, C), (B, C, A), (A, C, B), (C, A, B) e (C, B, A). Agora, se você tiver 5 itens, poderá ter 120 ordenações diferentes, se tiver 10 itens poderá ter 3.628.800 e se tiver 20 itens poderá ter 2.432.902.010.000.000.000 ordenações diferentes. Isso mesmo, é um calculo fatorial, como nas corridas de cavalos.

Um dos principais benefícios dos métodos ágeis e do desenvolvimento em iterações é a redução do tempo das entregas para o cliente. Durante o planejamento de *releases* é importante procurar potencializar o *time-to-market*, por isso devemos descobrir qual escopo necessário para agregar algum valor ao cliente e, então, definir um release para entrega. Quanto menores forem os ciclos de release, mais rápido o cliente poderá se beneficiar do produto, e oferecer feedback para que ele possa ser melhorado.

Para traçar um planejamento de *releases* é preciso estabelecer períodos fixos de entrega. Recomenda-se que seja um curto espaço de tempo. Com a visão do produto em mente é necessário estabelecer quais são as funcionalidades que podem agregar maior valor, ou seja, as que possuem maior valor de negócio e serão incluídas mais rapidamente nos *releases*.

Tendo em mãos a velocidade do time é possível ter uma ideia de quantas funcionalidades podem ser incluídas em um determinado período de tempo, e então dividir as funcionalidades em *releases*.

Não planeje para prazos muito longos. Quanto mais longo o prazo, maior a incerteza. Não é necessário, e nem recomendável, escrever e analisar todas as histórias no início do projeto. Da mesma forma que a equipe aprende e aprimora suas técnicas de desenvolvimento a cada *release*, o cliente utiliza o software e novas ideias surgem, bem como ideias antigas podem ser deixadas de lado. É por isso que não é interessante tentar prever futuros muito distantes: quanto mais distante, maior a incerteza.

Uma grande diferença dos métodos ágeis em relação aos tradicionais é que os métodos ágeis não tentam impedir o cliente de realizar mudanças no planejamento.

Em vez disso, considera-se que o cliente também aprende ao longo do projeto e suas mudanças trarão benefício. É claro que deve haver algum tipo de proteção para que as mudanças com frequência exagerada não tornem o projeto inviável. Tanto o XP como Scrum aconselham que ao menos a iteração atual não seja modificada.

É importante entender a diferença e a relação entre iteração e *release*. Os *releases* representam entregas e as iterações representam ciclos que estão contidos dentro dos *releases*. As iterações representam um determinado intervalo de tempo em que algumas histórias serão implementadas.

Nas iterações podem estar contidas diversas atividades como as reuniões de revisão e retrospectivas, discutidas. É possível, por exemplo, definir *releases* mensais com quatro iterações de uma semana cada. Não existe regra quanto à periodicidade dos *releases*, afinal cada projeto tem suas particularidades e um *release* pode implicar em uma série de fatores, como por exemplo, implantação e treinamento. Por isso, o número de *releases* e iterações estipulados devem estar alinhados às necessidades da sua organização.

Ao final do *release* o software é entregue para o cliente pronto para produção.

3.11 Roadmap do Produto

O Roadmap é uma ferramenta útil para traçar um plano de alto nível sobre futuros *releases* e marcos importantes do produto que serão alcançados ao longo do tempo.

O Roadmap oferece visibilidade de que funcionalidades serão entregues primeiro e quais serão entregues depois e em quais *releases*, e além disso apresenta marcos importantes do produto que está sendo construindo (ver figura 3.8).
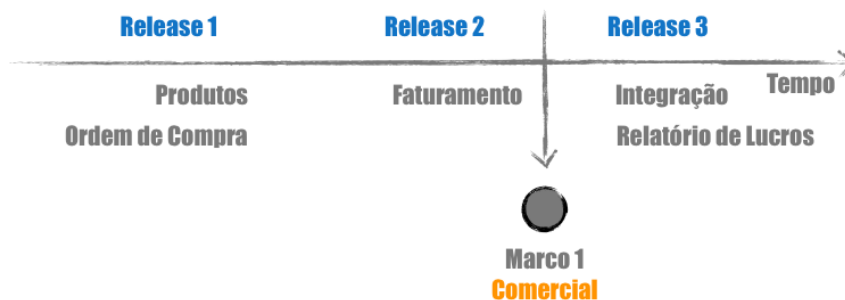
Figura 3.8: Roadmap

Como desenvolvimento de software é altamente complexo por natureza e difícil de se estimar e prever, num contexto ágil, o roadmap é uma ferramenta útil para mostrar intenções de *release* e apresentar marcos do produto, mas não tanto para definir prazos exatos de entrega de funcionalidades, justamente porque é natural que mudanças no escopo do produto ocorram ao longo do desenvolvimento. Por isso usa-se mais como um critério relativo do que absoluto, por exemplo, na figura 3.8 pode-se ver que a funcionalidade de preço, será lançada depois da de pedido, mas se está definindo uma data exata para isso.

Muitas vezes o roadmap pode ser útil para equipes de Marketing e de Negócio se posicionarem em relação a (mais ou menos) quando determinadas funcionalidades estarão disponíveis para fins de publicidade, treinamentos etc.

3.12 Mantenha as Opções Abertas

Um do princípios por detrás de muitas práticas dos métodos ágeis é a "liberdade de escolha" [40]. Chris Matts e Olav Maassen chamam-no de Opções Reais (em inglês *Real Options*).

Apesar de não ser exatamente a mesma coisa do que as opções do mercado financeiro (stock options), há uma relação entre os conceitos. As opções do mercado financeiro conferem ao titular o direito (e não obrigação/compromisso) de comprar um determinado ativo (ação, título ou bem qualquer) por um valor determinado, enquanto o vendedor é obrigado a concluir a transação.

Quando você compra um ingresso para o cinema, por exemplo, para você é uma opção porque mesmo tendo comprado o ingresso você pode ou não ir ver o filme; já para o cinema isso é mais um compromisso, porque ele já te vendeu o bilhete e agora precisará passar o filme no horário determinado. Vai depender do ponto de vista de cada um dos lados da relação.

Muitas vezes nós vemos e encaramos as coisas como compromissos em situações que poderíamos encarar como opções. Opções que podemos ou não exercer.

O pensamento de opções reais consiste em enxergar e avaliar todas essas opções ao seu redor e comprometer apenas de forma deliberada. Caso contrário, sempre se deve deixar para assumir o compromisso no último momento possível, ou seja, quando você tem as informações que precisa para tomar a decisão de comprometer-se de verdade.

Uma opção é uma escolha, uma decisão de escolher uma estratégia em vez da outra, de escolher um caminho em vez do outro, de comprar um produto em vez do

outro, de trabalhar em uma funcionalidade em vez de em outra. Cada uma dessas escolhas tem um retorno sobre o investimento. O desafio é trabalhar com a opção que tiver o melhor retorno sobre o investimento.

A diferença principal entre compromissos e opções é que podemos mudar de escolha de opções sem custo algum, ou com um baixo custo, mas mudar um compromisso, geralmente, gera custos ou problemas caso ele não seja cumprido. Temos o custo da opção e o retorno sobre o investimento que cada uma oferecer.

O mais interessante é que há valor nas opções que conhecemos e naquelas que não conhecemos também. Por isso é sempre importante considerar todas as opções possíveis e procurar aprender sobre opções até então desconhecidas antes de se comprometer com uma determinada opção.

Decidir no último momento possível (não confundir com negligenciar) é um dos princípios de Lean Software Development, e também é algo que faz parte de uma série de práticas utilizadas em equipes ágeis como Refactoring e Design Emergente (o design não é definido de antemão mas vai evoluindo ao longo do desenvolvimento), Planejamento Iterativo (a equipe se compromete apenas com as histórias de usuário de um iteração e o resto do backlog torna-se uma opção para o *Product Owner*; essas histórias podem ou não se tornar uma obrigação se incluídas no planejamento da próxima iteração), Incrementos Potencialmente Entregáveis (O PO pode exercer ou não a opção de fazer um *release* do resultado de uma iteração) etc.

Já com métodos tradicionais, o cliente precisa decidir sobre todo o escopo de antemão, e se precisar de alterações terá custos extras, por isso, é levado a tomar uma série de (más) decisões antecipadas, e esse mesmo erro permanece no Big Requirements Up Front e no Big Design Up Front.

Muitas pessoas preferem estar erradas do que estar em dúvida. Por isso, acabam tomando decisões ruins, por serem precipitadas [30]. Opções Reais nos apresentam uma melhor forma de lidar com nossas decisões.

Três regras das Opções Reais:

1. **Opções são valiosas**: Opções são valiosas porque podem ser executadas. No exemplo do ingresso do cinema, para o comprador é uma opção que está justamente na possibilidade de entrar na sessão para ver o filme. O valor está em ter opções em poder escolher.

2. **Opções expiram**: Opções têm um tempo de vida, no caso do ticket de cinema, ele só vale durante o tempo do filme, após a sessão do filme, a opção perde seu valor, independente de ser sido exercida ou não.

3. **Nunca se comprometa cedo, a menos que saiba o porquê**: Quando você se

compromete com uma opção está abrindo mão de todas as outras e do valor que elas podem oferecer. Quando nos comprometemos cedo demais, há um grande risco de não conhecermos todas as opções ou não escolhermos a melhor.

Injeção de Funcionalidades

Muitos projetos começam com uma grande lista de funcionalidades abstratas a serem desenvolvidas e, então, o time precisa correr atrás de um valor de negócio pouco claro e desconhecido. Ou seja, é como se soubessem que têm que correr, mas não soubessem a direção e nem por que estão correndo.

Quando alguém lhe pede para alterar algo da interface de usuário, por exemplo, qual é o valor que está por trás dessa mudança? Reduzir o tempo do Processo? Ganhar novos Usuários? Reduzir a chance de que um erro aconteça? Quando o cliente diz "Preciso que seja incluído o Campo X no Sistema", você sabe o motivo? Que valor está por trás disso? Será que isso realmente vai agregar algum valor? Para quem? De que forma?

Às vezes o PO dá ênfase no "como" em vez de enfatizar "o porquê". Mas se isso for invertido, e o PO trouxer o porquê, junto com o time, podem chegar a um "como" muito melhor.

A Injeção de Funcionalidades é um Processo de Análise de Negócios criado por Chris Matts com base na Teoria das Opções Reais para resolver esse problema. O processo possui três passos bem simples.

1. Persiga o Valor de Negócio

Quando o valor de negócio não está claro, as equipes não podem buscar formas diferentes e mais eficientes de se conseguir o valor. A funcionalidade pode ser entregue, porém de forma distorcida e não levar o valor que era esperado.

A Injeção de Funcionalidades (em inglês Feature Injection) propõe que tudo comece com um modelo de entrega de valor [6], como por exemplo: "Nós esperamos que o tempo médio de faturamento reduza de 20 minutos para 5 minutos, que o número de colaboradores dedicados ao processo reduza de 100 para 40, e a quantidade de emissões incorretas reduza de 900 notas fiscais para 100 notas fiscais."

Um modelo como esse permite que o time avalie as funcionalidades que estão desenvolvendo e se, de fato, poderão ou não contribuir para que o modelo de valor seja colocado em prática. Torná-lo explícito ajuda a criar um entendimento compartilhado de onde o valor está vindo. Com um objetivo claro, também fica mais fácil para se definir o que entra ou não entra no escopo e qual a prioridade de cada

item do backlog. Basta procurar entender em que a funcionalidade contribuirá com o modelo de valor.

2. Injete as Funcionalidades

Depois de ter o modelo bem definido, é hora de criar uma lista de funcionalidades que deverão ser desenvolvidas para atingir o valor de negócio. O importante é que deve haver uma ligação clara e objetiva entre as funcionalidades e valor de negócio [6].

Muitos *Product Owners* começam perguntando "o que" nós precisamos, em vez perguntar "para que" nós precisamos, e isso acaba levando a uma série de funcionalidades que para nada contribuem. Por isso, com Injeção de Funcionalidades você começa de trás para frente, ou seja, em vez de perguntar o que eu preciso (as entradas do sistemas) você pensa no porquê de precisar (nas saídas do sistema).

É importante lembrar que o valor não está nas funcionalidades em si, mas no resultado que o uso delas fornece ao usuário.

Tradicionalmente, as pessoas de negócio vêm com soluções meio prontas em vez de apresentar o valor de negócio que estão buscando. Com Injeção de Funcionalidades o time propõe soluções em cima do valor apresentado.

De certa forma, a Injeção de Funcionalidades é semelhante a TDD (Desenvolvimento Guiado por Testes), porque em vez de começar por escrever o código da funcionalidade, no TDD, você começa fazendo um teste que verifica se o resultado está correto e depois faz a implementação da funcionalidade. Com Injeção de Funcionalidades, você começa com o Output do Sistema (Saídas, Resultado) e depois descobre os Inputs (Entradas) de que precisa para conseguir o Output.

Para dar ainda mais ênfase no valor de negócio, Cris Matts, propõe uma inversão no formato tradicional das histórias de usuário de Mike Cohn:

Como um <papel>

Eu quero <funcionalidade>

para que <valor de negócio>

A injeção de Funcionalidades começa pelo valor de negócio:

Para que <valor de negócio>

como um <papel>

eu quero <funcionalidade>

Exemplo de Injeção de Funcionalidade

Para aumentar o número de visitantes no meu site como um publicitário (e aqui entram as opções)

Para aumentar o número de visitantes no meu site como um publicitário eu quero integrar nosso site com o Facebook

Para aumentar o número de visitantes no meu site como um publicitário eu quero publicar notícias diariamente

Para aumentar o número de visitantes no meu site como um publicitário eu quero otimizar o site para mecanismos de buscas

Note que o valor permanece, mas a funcionalidade pode mudar. Por isso é importante que o time foque em obter o valor de negócio e não a funcionalidade.

3. Encontre Exemplos

Agora é preciso encontrar as variáveis que podem afetar o resultado, ou seja, expandir o escopo e com a Injeção de Funcionalidades. Isso é feito através de exemplos: trabalhar com exemplos é uma ótima forma de verificar se as premissas dos requisitos são de fato válidas.

Comece com os exemplos mais simples, depois parta para os mais específicos e complicados. Busque exemplos específicos e não generalizados.

Procure por cenários, por casos reais. É horas de buscar os "quandos", os "ses", os "naquele caso", os "em particular" etc. Peça exemplos reais para as pessoas de negócio.

A Injeção de Funcionalidade permite que você não se comprometa cedo demais como uma solução ruim que pode nem sequer agregar valor, mas que, em vez disso, o valor de negócio seja o foco do processo e que puxe tudo o mais que for preciso para que ele possa ser conquistado.

3.13 E agora, o que eu faço amanhã?

Você se responsabiliza ativamente pelo resultado do seu time, ou apenas por suas contribuições individuais? Reflita sobre isso.

A visão do produto que você está ajudando a construir está disseminada entre todos os membros da sua equipe? Que tal criar um discurso de elevador para seu produto?

Quais são os marcos mais importantes do seu produto? Faça um roadmap de seu produto e identifique-os.

Quais os diferentes perfis de usuários que utilizam ou utilizarão o produto que você está ajudando a criar? Discuta com sua equipe e crie algumas personas, procure associá-las com as funcionalidades que vocês estão desenvolvendo atualmente.

Qual é a frequência com que vocês entregam software para o cliente? Essa frequência poderia ser diminuída de alguma forma?

No próximo planejamento da sua equipe, proponha fazer um mapeamento das histórias de usuário, depois de escrever as histórias e organizá-las verifique se estão atendendo os critérios INVEST e discuta sobre isso com a equipe.

Sua equipe já está utilizando limites de trabalho em progresso (WIP)? Se não, que tal conversar com time e definir alguns limites iniciais para identificar gargalos e problemas de trabalho parado?

Seu time está mantendo as opções abertas? No que você estão se comprometendo cedo de demais sem saber o porquê?

Você e seu time sabem o qual é o valor de negócio que está por trás de cada uma das funcionalidades que estão desenvolvendo? Será que vocês podem ajudar o PO a encontrar formas mais eficientes e baratas se conquistar o valor de negócio que está sendo perseguido?

Capítulo 4

Entregando Valor

Este é o segundo nível de fluência ágil, no qual os benefícios predominantes são a alta produtividade e a melhoria na qualidade externa do produto.

É um momento oportuno para investimento nas capacidades técnicas da equipe e, por isso, agora serão apresentadas algumas técnicas e práticas de engenharia utilizadas por equipes ágeis.

Equipes neste nível de fluência têm a habilidade de manter seus produtos sempre (ou quase sempre) em um estado de pronto-para-entrega, permitindo assim que o produto seja atualizado com a velocidade e frequência que o mercado exige.

Capacitar uma equipe para alcançar um alto nível de habilidade técnica requer tempo, esforço e muita prática. Cursos, treinamentos, contratação de desenvolvedores experientes para ensinar os menos experientes podem ser importantes para se atingir esse objetivo [57].

O aprendizado de técnicas mais avançadas e o pagamento da dívida técnica criada pela falta de experiência e código legado podem reduzir a produtividade neste momento. Mas apesar do alto investimento, uma equipe bem capacitada tecnica-

mente poderá criar produtos com menos defeitos, o que resultará em mais tempo para produzir funcionalidades que realmente possam agregar valor.

Algumas práticas predominantes neste estágio são: programação em par, integração contínua, propriedade coletiva de código, e desenvolvimento orientado a testes (TDD).

A seguir faremos um rápido estudo sobre cada uma dessas práticas ágeis.

4.1 Testes Ágeis

Métodos ágeis defendem a ideia de que é preciso criar software de qualidade desde o início, ao invés de assegurar a qualidade apenas no final do processo.

Novas funcionalidades serão entregues com alta frequência e não se pode permitir que as alterações realizadas no software para criar as novas funcionalidades façam com que funcionalidades já existentes deixem de funcionar. É para prevenir que isso aconteça que existem os testes de regressão que podem ser realizados manualmente ou automatizados.

Além disso, deve-se prevenir defeitos a todo o momento, porque quanto antes um defeito for detectado, mais rápido e barato será para resolvê-lo.

O que é mais barato: descobrir que uma alteração em uma parte do software criou um defeito em outra através do feedback de um testes de unidade durante o desenvolvimento e então resolver o problema na mesma hora, enquanto ainda se está com as regras de negócio em mente; ou descobrir posteriormente que o defeito existe porque não passou pela qualidade ou, ainda pior, descobrir que o problema existe apenas quando o software já estiver em produção?

A resposta, claro, é muito óbvia: quanto mais cedo você descobrir que o problema existe, mais rápido e barato será para resolvê-lo. A mesma ideia é aplicada na Toyota para defeitos em carros.

Ao longo desta seção, estudaremos diversas técnicas e abordagens diferentes para testes que, combinadas, podem ser úteis e importantes para o desenvolver software com qualidade.

A pirâmide de testes, criada por Mike Cohn [20] é um modelo interessante que além de defender que diferentes tipos de testes podem ser combinados, também sugere uma proporção dentre os diferentes tipos, como pode ser visto na figura 4.1:
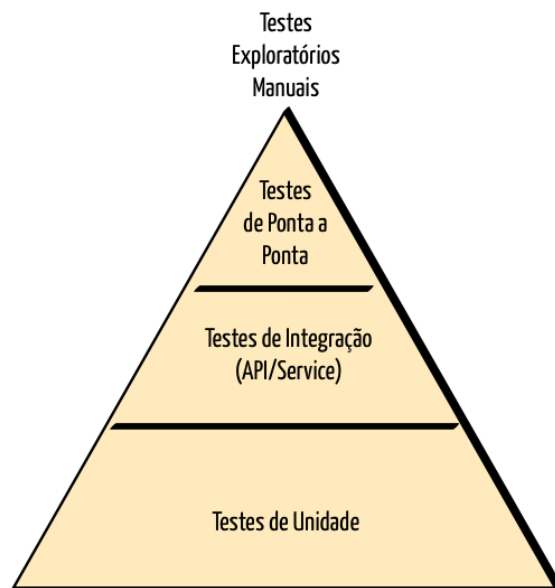
Figura 4.1: Pirâmide de Automação de Testes

Diferentes tipos de testes têm objetivos distintos e seu custo de automação e tempo de execução também diferem.

Testes de Unidade, geralmente, são os mais rápidos se de automatizar e também são rápidos de executar, por isso, eles estão na base da pirâmide e representam a maior parte dos testes propostos pelo modelo.

Em seguida, temos os testes de integração (também chamados de testes de serviço ou testes de API). Eles são mais completos e, geralmente, combinam diferentes componentes que trabalham em conjunto para realizar em um determinado sistema. São realizados diretamente na aplicação sem passar pela interface de usuário.

Em terceiro lugar, e no topo da pirâmide, temos os testes de ponta a ponta (também chamados de interface de usuário, testes de sistema, ou testes de aceitação) que, como o nome sugere, testam a aplicação desde a interface do usuário, por isso são os mais completos e, geralmente, são também os mais caros de se automatizar, e os mais lentos para se executar.

É importante analisar os custos e benefícios de cada um dos tipos de testes propostos na pirâmide para se fazer bons investimentos em automação de testes.

Há também os testes exploratórios, que são executados sempre manualmente

por um membro do time, com a finalidade de ir além de onde os testes automatizados puderam chegar. Eles não seguem um script pré-planejado e se beneficiam da capacidade de investigação, da observação, da analise e da adaptação do testador. Através deles os testadores buscam e encontram riscos que extrapolam o universo de riscos que puderam ser previstos de antemão pela equipe.

Em se falando de prevenção de defeitos de software, TDD (Test-Driven Development) é uma ótima ferramenta na maioria dos contextos. Essa técnica sugere a escrita de testes de unidade antes mesmo de se codificar a funcionalidade a ser testada. Com isso, é possível garantir alta cobertura de testes [7] e, consequentemente, feedback rápido em caso de alterações que possam gerar efeitos colaterais quebrando outras funcionalidades.

A seguir exploraremos um pouco melhor cada um dos diferentes tipos de testes. É importante ter em mente que nenhuma abordagem é melhor do que a outra; na verdade, cada um deles é uma boa ferramentas para resolver certos tipos de problema e, você, provavelmente, precisará combinar todos para garantir a qualidade necessária no produto que está desenvolvendo.

Testes de Unidade

Escrever testes de unidade é uma prática essencial para se garantir a qualidade de um software. Um desenvolvedor ágil evita ao máximo escrever código sem escrever também um teste de unidade para verificar se tudo está funcionando da forma como deveria, por isso, no método XP, utiliza-se uma prática conhecida como TDD - Test-Driven Development ou Desenvolvimento Guiado por Testes.

Com TDD os testes de unidade são escritos antes mesmo do que as classes e métodos a serem testados, melhorando assim a garantia de que todo o código está coberto por testes. É importante ressaltar que o conceito de TDD vai muito além de apenas escrever testes de unidade, e apresenta, na verdade, uma forma inovadora de se pensar no design de um software.

É muito raro que um desenvolvedor acredite que possa escrever código que sempre funcionará na primeira vez. Na verdade, muitos ficam surpreendidos quando isso acontece. Por isso, independente de escrever testes antes ou depois, é preciso testar para garantir que o software está funcionado da maneira que deveria funcionar.

Uma alteração em um determinado trecho de código de uma parte de um sistema pode causar um problema em outra parte, e os componentes geralmente contêm dependências, que quando alteradas podem modificar seus comportamentos de forma

inesperada: são os conhecidos efeitos colaterais.

Além de ajudar na melhoria da qualidade, os testes de unidade podem ajudar o desenvolvedor a entender melhor as regras de negócio do sistema, uma vez que, provavelmente, haverá um teste para cada regra. Dessa forma, eles poderão ser considerados também uma "documentação viva", porque além servir como referência para compreensão das regras de negócio, servirão também para garantir que essas regras estejam implementadas corretamente e que o software esteja se comportando da maneira que se espera.

É importante que os testes sejam fáceis de se executar, porque pouco adiantará ter testes de unidade que nunca são executados ou que são executados com uma frequência muito baixa.

Testando antes com Desenvolvimento Guiado por Testes

Desenvolvimento Guiado por Testes (TDD) é um processo em que o desenvolvedor começa a implementação de uma nova funcionalidade pelo teste e deve, o tempo todo, fazer de tudo para que seu código fique simples e com qualidade [7].

TDD é realizado através de um ciclo com os seguintes passos:
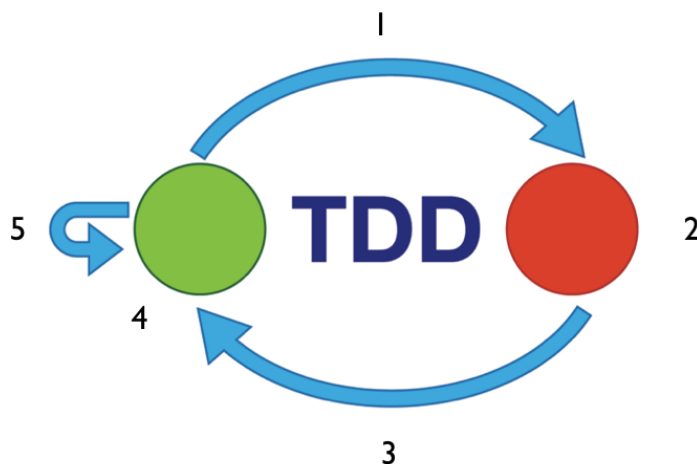
Figura 4.2: Ciclo de TDD (Imagem de Maurício Aniche)

1) Adicione um teste rapidamente.

2) Execute todos os testes e observe o novo teste falhar.

3) Faça uma pequena mudança para fazer o teste passar .

4) Execute todos os teste e observe que foram bem sucedidos.

5) Refatore.

O uso de TDD assegura que todo código adicionado ao repositório seja testado e, além disso, dá ênfase em se escrever apenas o código necessário para que os testes passem, evitando que o desenvolvedor escreva mais do que é necessário, contribuindo para um código mais simples, enxuto e também mais fácil de se dar manutenção.

Testando de Ponta a ponta com testes de Aceitação

Testes de Unidade não são os únicos tipos de teste que devem ser realizados. Existem outros tipos que, unidos aos unitários, poderão trazer uma qualidade ainda melhor ao software. É o caso dos testes de aceitação, também conhecidos como testes funcionais ou testes de usuário final.

Testes de aceitação, diferente dos testes de unidade que verificam apenas o comportamento de uma classe ou método, testam o comportamento do sistema de uma forma mais ampla. Geralmente, representam o contexto completo de uma história de usuário, ou caso de uso. Os testes de aceitação, são baseados em critérios de aceitação relacionados com regras de negócio que podem ser especificadas pelo *Product Owner*.

É possível realizar testes de aceitação manualmente ou automatizá-los. A realização de testes manuais pode reduzir drasticamente a velocidade do time, por isso, recomenda-se a automação dos testes.

Algumas ferramentas como o Selenium, por exemplo, permitem realizar a automação de testes de aceitação em aplicações web, gravando ações do usuário no navegador para que o cenário possa ser reproduzido e os resultados verificados posteriormente.

Vá além de testes exploratórios

Testes Manuais são testes realizados por pessoas, geralmente, que têm o papel de testadores em um time de desenvolvimento de software. Alguns testes manuais consistem basicamente na execução de um script escrito de antemão que possui uma sequência de passos a serem seguidos e um resultado a ser validado. Esse tipo de teste pode ser facilmente automatizados.

Há, porém, um segundo tipo de teste que é realizado manualmente, conhecido como **testes exploratórios**. Eles se aproveitam de todos o potencial do testador para investigar e descobrir cenários, e possivelmente defeitos ou efeitos colaterais, que não haviam sido previstos anteriormente.

Para este tipo de testes não há um script preestabelecido e o testador vai muito além do óbvio criando e adaptando sua abordagem ao longo do processo.

Lidando com Código Legado sem Cobertura de Testes

É muito comum que uma equipe que não trabalhava com métodos ágeis e passa por uma transição se depare com um projeto para evoluir que não possui testes automizados porque na cultura anterior testar e automatizar não era uma prática disseminada dentre os membros do time.

Fazer alterações em um sistema que não possui um bom conjunto de testes automizados é muito difícil. O maior risco é que você pode facilmente quebrar alguma funcionalidade sem que ninguém perceba.

Por isso Michael Feathers e muitos outros agilistas consideram que código sem cobertura de testes é código legado, não importa se foi escrito oito anos ou oito horas atrás [24].

Uma das opções para mudar essa cenário é melhorar a cobertura de testes pouco a pouco a cada iteração. Para isso, Henrik Kniberg sugere o seguinte processo [35]:

1) Liste seus casos de testes.

2) Classifique cada teste por risco, o custo de se testar manualmente e o esforço para automatizar o teste (veja na figura 4.3.

3) Ordene por prioridade.

4) Automatize alguns testes a cada iteração, começando pelos casos de maior prioridade.

Caso de Teste	Risco	Custo de Testar na Mão	Custo de Automizar o Teste
Emitir Nota Fiscal	Alto	2 horas	20 pontos
Cancelar Nota Fiscal	Alto	1 hora	8 pontos
Realizar Pagamento	Alto	3 hora	12 pontos
Cadastrar Produto	Médio	2 horas	4 pontos
Excluir Produto	Baixo	1/2 hora	2 pontos
Incluir Conta Bancária	Baixo	1/2 hora	2 pontos

Figura 4.3: Lista de Casos de Teste

Uma vez tendo a lista pronta e classificada é possível priorizar e começar pelos testes cuja falta apresenta maior risco, e que possuem maior custo de realização manual e menor custo de automação.

Quando não se tem testes automatizados a única forma de garantir que uma alteração não quebrou uma funcionalidade preexistente é realizando testes manuais. Não é incomum que o custo de realizar os testes manualmente seja maior do que o custo de automatizar, por isso pode ser interessante estimar e apresentar aos interessados no projeto para que possam entender o custo benefício do investimento na automação deles.

Nem sempre temos a oportunidade de trabalhar em novos projetos em que podemos garantir alta cobertura de teste desde o princípio mas isso não é motivo para desanimar; pouco a pouco, iteração a iteração é possível mudar a realidade do projeto e melhorar a qualidade, aumentando a cobertura de testes.

4.2 Simplificando o Código com Refatoração

Complicar é fácil. Difícil é simplificar
– Max Gehringer

É natural que ao longo do tempo a base de código de um sistema vá se tornando

cada vez maior, assim como o número de funcionalidades disponíveis e a complexidade do código fonte.

Sem boas práticas para manter o código limpo e organizado, à medida que mais código vai sendo acrescentado, mais desorganizado e difícil de se compreender o repositório vai se tornando. Isso impacta negativamente na produtividade dos desenvolvedores que encontrarão mais dificuldade em dar manutenção nas funcionalidades existentes e em adicionar novas funcionalidades.

Qualquer idiota pode escrever código que um computador pode entender. Bons programadores escrevem código que seres humanos podem entender.
– Martin Fowler

Refatorar é alterar a estrutura do código sem alterar o seu comportamento. É uma prática que permite que o desenvolvedor melhore o design do código, tornando-o mais limpo e fácil de se compreender. É uma técnica essencial para prevenir que o código se deteriore [20].

Mas para que o desenvolvedor possa alterar o código com segurança de que o comportamento não será alterado, é essencial que o código possua uma boa base de testes que atuem como uma rede de segurança prevenindo que o comportamento do software perca a consistência. Por isso, escrever testes e refatorar são técnicas que, geralmente, são utilizadas em conjunto e é exatamente por isso que ambas são utilizadas no desenvolvimento guiado por testes, conforme estudaremos a seguir.

Um bom design evolui ao longo da vida de um sistema, mas Poppendieck afirma que um bom design não acontece por acidente. Por isso, ao encontrar algo errado na base do código, é preciso parar de incluir novos comportamentos ao software e resolver o problema.

Para garantir uma boa qualidade de código é preciso refatorar. À medida que novos comportamentos são incluídos no software, por consequência aumentando a complexidade, existe uma grande tendência de se escrever código duplicado e torná-lo mais difícil de compreender e de dar manutenção.

Por outro lado, é preciso tomar cuidado para não cair em um perfeccionismo extremo e gastar tempo tentando aperfeiçoar detalhes sem importância. Para facilitar a identificação do momento correto para realizar a refatoração, Poppendieck cita cinco características de um sistema íntegro. Se qualquer uma destas estiver faltando, é sinal de que é hora de refatorar. São elas:

1. Simplicidade: Na maioria das vezes um design simples é o melhor design.

Design Patterns, quando bem aplicados, são uma ótima ferramenta para aplicar soluções simples a problemas complexos;

2. Clareza: O código fonte deve ser facilmente compreendido por todos os membros da equipe. Todos os elementos do código devem ser devidamente nomeados para que sejam facilmente compreendidos por todos;

3. Eficácia: Um bom design deve produzir o resultado correto, isto é, deve atingir o objetivo pelo qual foi criado;

4. Sem repetição: O mesmo código não deve existir em dois lugares diferentes. Quando uma mudança precisa ser feita em muitos lugares o risco de defeitos cresce exponencialmente;

5. Utilidade: Toda funcionalidade de um sistema precisa ser útil para seus usuários. Quando uma funcionalidade deixa de ser necessária, cria-se desperdício em mantê-la; em muitos casos é melhor eliminá-la.

4.3 Código Limpo

Robert C. Martin, um dos criadores do Manifesto Ágil, conhecido na comunidade como Uncle Bob, defende com afinco a ideia de que, ao longo do tempo, a base de código de um projeto vai se tornando mais bagunçada, e difícil de se entender, impactando fortemente na produtividade da equipe, que tende a zero, ou seja, a ponto de em um determinado momento a equipe não conseguir mais evoluir e incluir novas funcionalidades no software [41].

De fato, muitos softwares, chegam a um tal ponto que precisam ser reescritos do zero para que possam continuar a ser evoluídos.

Algumas características de código limpo são:

- Elegante, fácil de se compreender, e agradável de se ler.
- Simples e direto.
- Segue princípios de programação.
- Sem duplicidade.
- Bem testado.
- Parâmetros, Métodos, Funções, Classes e Variáveis possuem nomes significativos.

- Código é autoexplicativo (sem necessidade de comentários para compreender o código).
- Código bem formatado (é possível ler o código sem precisar utilizar a barra de rolagem).
- Métodos e Classes devem ter uma única responsabilidade (princípio da responsabilidade única).

Trabalhar para manter o código sempre limpo é, além de uma atitude profissional do desenvolvedor, muito eficiente em termos de custos para a organização, uma vez que a mantenabilidade do software será sustentável a médio e longo prazo.

Para Uncle Bob, desenvolvedores profissionais escrevem código como profissionais, e código escrito por profissionais é código limpo!

4.4 Propriedade Coletiva do Código

A Propriedade Coletiva do Código é uma das práticas de XP e consiste na ideia de que o time é responsável por todo o repositório de código em vez de os indivíduos serem responsáveis apenas pelo código que eles mesmos escreveram.

Essa prática faz com que manter e garantir a qualidade do código seja responsabilidade de todos os membros do time. Qualquer um pode realizar as alterações que forem necessárias em qualquer parte do sistema [66].

Nessa abordagem, quando um desenvolvedor precisar alterar determinada parte do código, não precisará pedir permissão para o desenvolvedor que o escreveu e terá total autonomia de fazer o que precisa ser feito.

Essa prática funciona muito melhor quando em conjunto com revisões de código, programação em par e integração contínua. Além de melhorar a qualidade, também ajuda na eliminação de ilhas de conhecimento.

Você já viu uma situação em que determinada parte de um projeto precisa ser alterada mas a única pessoa do time que escreveu, compreende e conhece aquele código está de férias? Esta pessoa é um ilha de conhecimento, e essa situação é nociva para o projeto, porque se cria uma dependência muito grande em uma única pessoa, o que possivelmente também resultará em gargalos e sobrecarga.

Código Coletivo é uma boa vacina contra ilhas de conhecimento, e reduz o risco se alguém, por alguma razão, deixar o projeto.

Nessa abordagem de propriedade de código, se um desenvolvedor encontrar um código pouco claro, sem testes, ou fora dos padrões e qualidade, não importa quem o escreveu, esse código poderá e deverá ser melhorado.

4.5 Linguagem ubíqua

Desenvolvedor: - "Nós estamos com um problema no NFBusinessDelegate quando enviamos o objeto para o Service, parece que a transação não é iniciada e o DAO e, então, não funciona bem."

Cliente: - ?

Desenvolvedores devem falar a linguagem de negócio [66].

Um dos grandes desafios do desenvolvimento de software é, muitas vezes, a comunicação entre desenvolvedores que não são **experts `no domínio (áreas de negócio) do software com os` experts** do negócio. [66].

Em seu livro Domain-driven design (DDD), Eric Evans cunhou o termo Linguagem Ubíqua que diz respeito à criação de uma linguagem comum entre experts no negócio e o time de desenvolvimento de software. Dessa forma, essa mesma linguagem de negócio é utilizada no próprio código fonte do software, na hora dar nomes a parâmetros, métodos, variáveis, classes, testes, etc.

É importante que o time compreenda cada um dos termos utilizados por clientes. É comum que os experts do negócio tenham seus próprios jargões, assim como desenvolvedores e testadores também têm seus próprios. Em um projeto sem uma linguagem comum, o tempo todo, desenvolvedores precisam traduzir seus termos para o expert de negócio, assim como esse também têm que traduzir seus termos para os desenvolvedores, e isso prejudica a comunicação [23].

Com uma linguagem comum a linguagem das discussões e do planejamento será a mesma utilizada no código. Se, em um sistema de logística, por exemplo, o cliente se refere a cargas, os desenvolvedores, no código fonte, utilizarão o mesmo o termo cargas, e mercadorias, ou produtos, ou lote, por exemplo.

O primeiro passo para melhorar a comunicação é reconhecer a sua fragilidade, e a linguagem ubíqua é uma excelente ferramenta para ajudar você e sua equipe a se comunicarem melhor com as pessoas de negócio envolvidas em seu projeto.

4.6 Design Ágil é Design Iterativo

Um dos maiores erros do desenvolvimento em cascata é achar que o conhecimento existe apenas na forma de requisitos levantados antes do início da implementação e

separados do desenvolvimento.

Em uma perspectiva ágil, o desenvolvimento de software é um processo de criação de conhecimento, e que na prática, o design detalhado de um software ocorre somente na codificação. Um design prematuro, ou seja realizado antes da codificação, não pode prever a complexidade encontrada durante a implementação, e tampouco pode utilizar-se do feedback que só pode ser recebido ao longo do desenvolvimento.

O design de um software deve evoluir ao longo de seu desenvolvimento. Tentar fixá-lo prematuramente resultará em desperdício.

Em se tratando de requisitos, deve-se seguir a mesma linha de pensamento, de forma que os requisitos devem ser detalhados apenas quando estiverem perto da hora de desenvolvê-los. É importante também estar preparado para as prováveis mudanças neles, uma vez que o cliente também está aprendendo e evoluindo suas ideias a respeito da solução que está sendo desenvolvida.

As abordagens prematuras de levantamento de requisitos e de design são chamadas, respectivamente, de Big Requirements Up Front (BRUP) e Big Design Up Front (BDUF), e têm sido amplamente abordadas na comunidade ágil como práticas a serem evitadas, pois o conhecimento não é gerado por antecipação, mas sim através de experimentação, codificação e *releases* frequentes que permitem receber feedback dos clientes. O desenvolvimento de software iterativo e incremental encaixa-se perfeitamente nessa linha de pensamento.

YAGNI: Você não vai precisar disso...

"Não pague o preço da complexidade a menos que você realmente precise."
– Neal Ford

O princípio YAGNI (*You Ain't Gonna Need It* ou Você não vai precisar disso) muito disseminado na comunidade ágil, defende a ideia de que você deve evitar especulações no desenvolvimento de software [25].

Isso ocorre naquelas situações em que se pensa "Eu acho que vamos precisar disso no futuro, então, vou aproveitar e já fazer". Dessa forma, adiciona-se complexidade desnecessariamente no projeto, e, quando mais complexidade, mais difícil de se alterar torna-se o software.

Procure fazer apenas aquilo que realmente é preciso agora!

4.7 Definindo o significado de Pronto

A Definição de Pronto (em inglês Definition of Done, referenciado com o acrônimo DoD), é, basicamente, um **checklist** do que deve ser feito antes que uma história possa ser considerada potencialmente entregável.

Não há um **checklist** oficial que deva ser usado por todas as equipes ágeis do mundo, porque, considerando a complexidade do desenvolvimento de software, cada equipe é única e está resolvendo um problema dentro de um contexto único. Por isso, cada equipe deverá descobrir o que deve fazer parte de sua definição de pronto de acordo com seu contexto ou seja, levando em consideração a natureza do produto, a tecnologia que está sendo utilizada, as necessidades dos usuários etc.

Alguns itens comuns são:

- Código refatorado.
- Código dentro dos padrões de codificação.
- Código revisado ou feito em par.
- Código integrado no sistema de controle de versão.
- Documentação de arquitetura atualizada.
- Testes de unidade realizados.
- Testes de aceitação realizados.
- Testes exploratórios realizados.
- Nenhum defeito conhecido pendente.
- PO aceitou na história.
- Manual do Usuário atualizado.

É importante que essa definição seja escrita e que fique disponível e visível para todos. Isso evita que, por exemplo, uma história de usuário seja dada como pronta, mas a documentação para o usuário final não tenha sido escrita ainda, e que, então, esse trabalho pendente seja empurrado de uma iteração para a outra, porém não seja considerado nas estimativas fazendo com que um falso senso de baixa produtividade permeie o ambiente.

Além disso, a definição também é importante para que haja um mesmo entendimento do significado de pronto para todos os membros do time, e para que nada de importante seja esquecido ou deixado de lado.

A mesma ideia pode ser aplicada para além das histórias de usuários: também para iterações e *releases*. Geralmente, entende-se que uma iteração está pronta ao fim de um intervalo de tempo predefinido, mas pode ser interessante que determinadas atividades sejam realizadas antes de se dar uma iteração como pronta, como por exemplo fazer *deploy* em um servidor de homologação ou algo do tipo.

A participação do PO, na construção dessa definição é muito importante, uma vez que ele conhece melhor do que ninguém as expectativas do cliente.

A definição de pronto não deve ser estática, ao contrário disso, é natural que, à medida que o time for evoluindo e amadurecendo, a definição de pronto seja alterada para atender critérios de mais e mais qualidade [59].

4.8 Integração Contínua

É natural que em um projeto de desenvolvimento de software os desenvolvedores do time trabalhem em paralelo no desenvolvimento de funcionalidades e correções. Muitas vezes, podem acabar por alterar as mesmos arquivos gerando conflitos, ou até mesmo, alterar arquivos diferentes, que depois de combinadas as alterações, o software apresente um comportamento inadequado. De qualquer forma, de tempos em tempos, as alterações precisarão ser integradas em uma base comum de código.

O processo de integração, geralmente, é visto por desenvolvedores como algo custoso e demorado: as peças funcionam bem quando separadas, porém, para tentar uni-las e fazê-las trabalhar em conjunto o cenário se torna mais complicado.

Manter o código sempre integrado e pronto para ser entregue é um grande desafio, e essa é uma das finalidades da integração contínua. A ideia da integração contínua é que todos os desenvolvedores realizem integração, isso é, sincronizem o código fonte produzido com o sistema de controle de versão, a maior quantidade de vezes possível ao longo do dia.

É recomendado que a cada integração todos os testes de unidade sejam executados para assegurar que o *build* não seja quebrado e que tudo está funcionando corretamente.

Chamamos de *build* (construção) o processo de gerar um pacote executável de software. Isso inclui a compilação de código fonte e, além disso, pode-se acoplar uma série de outros processos para garantir que o pacote está em boas condições, como

análise estática de código, verificação de padrões de codificação, execução de testes automatizados, etc. Alguns exemplos de ferramentas de build são: make, Gradle, ant, maven e BuildR.

Se o software possuir um bom nível de cobertura de testes, e os testes forem bem escritos, essa prática pode reduzir consideravelmente o número de defeitos em produção e garantir que o código fonte produzido para implementar novas funcionalidades não fez com que algo deixasse de funcionar como deveria.

Existem diversas ferramentas, na verdade servidores de integração contínua — inclusive alguns são de uso livre e open-source — que podem ajudar na integração contínua.

É possível configurar essas ferramentas para que cada vez que algum desenvolvedor enviar algum novo código fonte para o sistema de controle de versão, todos os testes sejam executados e o desenvolvedor receba uma notificação caso o **build** tenha sido quebrado porque, por exemplo, algum teste tenha falhado.

Muitos servidores de integração contínua também possuem extensões que permitem que sejam gerados relatórios de cobertura de testes, identificação de defeitos óbvios no código (através de ferramentas de análise estática), entre outras coisas interessantes.

Se você ainda não usa nenhuma ferramenta, eu sugiro que comece verificando o Jenkins (jenkins-ci.org) e o CruiseControl (cruisecontrol.sourceforge.net).

4.9 Programação em Par

A programação em par é uma técnica na qual dois programadores trabalham em um mesmo problema, ao mesmo tempo e em um mesmo computador. Enquanto uma pessoa (o condutor) assume o teclado e digita os comandos que farão parte do programa, a outra (o navegador) a acompanha fazendo um trabalho de estratégia [60].

A revisão de código, na programação em par, acontece em tempo real. Enquanto um desenvolvedor está codificando, o outro está revisando o código. Pequenos erros, que poderiam ser extramente difíceis de se corrigir posteriormente, podem ser facilmente identificados e corrigidos nesse momento.

Os antigos já diziam que "duas cabeças são melhores do que uma", e a programação também se beneficia disso. Dois desenvolvedores com um conjunto de conhecimento e experiência distintos são, em regra geral, capazes de resolver um problema

de forma muito mais eficiente do que poderiam fazer se estivessem trabalhando sozinhos. É o princípio da sinergia aplicado à programação.

Em um primeiro olhar, pode-se pensar que a programação em par é uma técnica que reduz a produtividade da equipe, e que se ambos os desenvolvedores estivessem trabalhando em paralelo resolveriam um determinado problema mais rapidamente do que trabalhando juntos.

A programação em par é a prática de desenvolvimento ágil de software com mais estudos científicos realizados na Academia até agora, e alguns desses estudos demonstram que, na prática, a programação em par não apresenta grandes diferenças de produtividade [67], e que, em geral, o produto é entregue com menos defeitos e com melhor qualidade, isso significa que menos tempo será desperdiçado em correções de defeitos e que a futura manutenção do software será mais fácil, ou seja, em uma análise de médio prazo, podemos concluir que a programação em par é, de fato, eficiente.

Um dos fatores que apoiam essa alta produtividade é pressão dos pares. Nestes tempos de redes sociais, e-mail, SMS, e diversas outras distrações, um desenvolvedor pode facilmente perder o foco e se distrair quando trabalha sozinho.

Trabalhando em par, o desenvolvedor assume um compromisso com seu colega reduzindo assim a distração e a perda de foco, e como há um colega acompanhando a codificação, há um preocupação maior em criar soluções com mais qualidade e elegância.

Programação em Par e Aprendizado

"Aprender é a única coisa de que a mente nunca se cansa, nunca tem medo e nunca se arrepende."
– Leonardo da Vinci

Trabalhando em par, o conhecimento do time será rapidamente disseminado entre os membros do time ajudando na eliminação de ilhas de conhecimento. Para que isso aconteça de forma eficiente, é importante que haja um revezamento constante entre os pares, de forma que todos tenham oportunidades de trabalhar com todos.

A Pirâmide dos Pares

Uma ferramenta interessante para incentivar o revezamento entre os pares e tonar visivel quem está pareando com quem ao longo de uma iteração é a Pirâmide dos Pares ou Pair-amid [44].

Trata-se basicamente de uma matriz que cruza todos os membros do time, e desconsidera os cruzamentos repetidos e os cruzamentos de indivíduo com ele mesmo, veja na figura 4.4.

	CAROL	ROBSON	LUIZ	ALINE
CAROL				
ROBSON	XXX			
LUIZ	X			
ALINE	XX	XXX	X	

Figura 4.4: Pirâmide dos Pares

No exemplo acima, podemos ver que até agora o Robson e o Luiz numca parearam, mas que o Robson pareou 3 vezes com Aline e vezes com a Carol.

Se a pirâmide dos pares ficar visível no quadro do time, tornar-se-á uma ferramenta útil para incentivar a programação em par e o revezamento dos pares resultando em maior compartilhamento de conhecimento entre os membros do time.

A programação em par cria um espaço para aprendizado contínuo dentro da equipe [60]. O trabalho em par também pode ser mais eficiente para disseminar conhecimento do que documentações uma vez que não se limita ao que está escrito e porque há troca de conhecimento entre ambas as partes.

É importante, porém, diferenciar programação e par de mentoria [1], a relação da programação não é de estudante e professor, mas em vez disso, é um relação de

duas pessoas trabalhando juntas. Se há uma diferença muito elevada no nível de conhecimento de dois profissionais trabalhando em par, provavelmente o que vai acontecer é que um deles irá ensinar o outro na maioria tempo, criando assim uma relação que caracterizaria mais mentoria do que com programação em par.

A mentoria é também, sem dúvida, uma prática muito válida e importante em uma equipe de desenvolvimento de software, mas é importante separar um conceito do outro, e estar ciente de que a mentoria, geralmente, não resulta na mesma produtividade da programação em par, e que, muitas vezes, é um investimento mais de longo prazo comparado com a programação em par.

Em suma, os principais benéficos da programação são melhor qualidade, equipe mais motivada, mais confiança e trabalho em equipe, disseminação de conhecimento, e aprendizado contínuo [33].

Mito: Sem Programação em Par, não há Agilidade

Martin Fowler, uma das personalidades mais respeitadas no mundo do desenvolvimento de software, escreveu um artigo em 2006 levantando alguns dos principais enganos sobre programação em par.

Fowler esclarece em seu artigo que "não é necessário programar em par para estar praticando um processo ágil" [29], nem mesmo se pode dizer que para aplicar Extreme Programming você é obrigado a programar em par. O máximo que se pode dizer é que alguém que quer aprender XP deve tentar programar em par e ver se funciona em seu caso em particular .

A programação em par não é nem mesmo citada no manifesto ágil, porém, é uma prática altamente recomendada para que se alcance os princípios ágeis.

4.10 Revisão de Código

A revisão de código consiste em um ou mais desenvolvedores lerem o código fonte com o objetivo de assegurar a qualidade do código e aprender no processo. Especialmente na implementação de histórias que não foram desenvolvidas com programação em par, recomenda-se, a prática de revisão de código.

A revisão de código melhora sua qualidade e reduz a taxa de defeitos [31]. As revisões são oportunidades não apenas de se prevenir erros no código, mas também

de compartilhar e disseminar o conhecimento entre os integrantes do time. Ao revisar o código alguém pode descobrir boas práticas que não conhecia, ou, em caso de encontrar alguma má pratica, então passa a ser uma boa oportunidade de discutir sobre formas melhores de solucionar o problema em questão.

Para que as revisões de código sejam de fato construtivas para o time, é importante manter uma atitude de trabalho em equipe quando um problema for encontrado e evitar encontrar culpados ou debochar daquele que cometeu o erro.

Em um ambiente ágil, parte-se do princípio de que todos estão fazendo o seu melhor, então, se eventualmente um problema for encontrado, procure se certificar de que todos saibam como evitar que o problema volte a ocorrer, e siga em frente.

As revisões de código tanto podem fazer parte da definição de pronto e, consequentemente, do processo, como podem ser esporádicas, sendo realizadas de tempos em tempos. Em algumas equipes um desenvolvedor revisa o código de outro, em outras equipes, a revisão é coletiva envolvendo várias pessoas.

Algumas organizações fazem uma mescla entre programação em par e revisão de código, de forma que tudo o que não for desenvolvido em pares, é revisado. Não há fórmula secreta para isso, é interessante testar diversos métodos e encontrar aquele que funciona melhor no contexto de sua equipe.

4.11 Dívida Técnica

Imagine que você é um desenvolvedor de software que está trabalhando em um sistema de transferências bancárias. Você está fazendo uma alteração que, teoricamente, seria muito simples, mas percebe que o design atual do sistema precisaria sofrer algumas alterações para que fosse possível criar testes de unidade para a alteração que está fazendo.

Você, agora, precisa tomar uma decisão: fazer um trabalho "rápido e sujo" para resolver o problema rapidamente, mas deixando o código sem testes, ou refatorar e alterar o *design* para que seja possível escrever testes de unidade.

Desenvolvedores precisam tomar decisões como essas diariamente e, a cada "trabalho rápido e sujo" que fazem, aumentam a dívida técnica do projeto.

Dívida técnica é uma metáfora criada por Ward Cunningham que ilustra as consequências desses *trade-offs*. A ideia básica por trás do termo é que diminuir a qualidade aumenta o tempo de desenvolvimento a longo prazo.

Para Cunningham, divida técnica inclui coisas que você escolhe não fazer agora, mas que impedirá o desenvolvimento futuro se deixado de lado [21]. Isso inclui,

protelação de refatoração, mas não inclui protelação de funcionalidade, a não em ser casos que a entrega foi "boa o suficiente" porém não satisfaz algum tipo de padrão de desenvolvimento.

Dívida Técnica ou Débito Técnico

No conteúdo sobre método ágeis da comunidade brasileira, especialmente em blogs, encontra-se muito o termo "débito técnico", mas entende-se que "dívida" é uma tradução mais correta para *debt* . Em inglês a palavra *debt* quer dizer **dívida** (um valor que não foi pago e deverá ser pago futuramente), já a palavra *debit* significa **débito** (um valor que foi retirado de uma conta bancária, por exemplo).

Depois da definição de Cunningham no início dos anos 90, a comunidade continuou a evoluir o conceito [53] e, atualmente, diversos "atalhos" tais como design ultrapassado, defeitos conhecidos que não foram resolvidos, baixa cobertura de testes de unidade, excesso de testes manuais, entre outros, também são considerados parte da dívida técnica.

Se ela for ignorada, em algum momento o código pode se tornar tão confuso e caótico que sua manutenção será prejudicada, aumentando o custo de desenvolvimento e suporte, diminuindo a produtividade do time, aumentando o número de defeitos devido à alta complexidade e dificuldade de compreensão do código, e finalmente diminuindo a motivação da equipe, devido à baixa produtividade e alto número de defeitos. É previsível que, em num cenário como esse, os clientes e usuários, provavelmente, também não estarão muitos contentes.

O valor da metáfora da dívida técnica está em tornar o time consciente desses problemas e de como a produtividade pode estar sendo afetada. Se o time fizer um acompanhamento da dívida técnica, é possível evitar que ela se torne cada vez maior e, em vez disso, trabalhar para pagá-la pouco a pouco.

É importante que a equipe mantenha um backlog de dívida técnica, ou seja, uma lista de tarefas que podem ser realizadas para melhorar o código, e essa lista pode ser priorizada pelo time e, então, pode-se negociar com o *Product Owner* para que pouco a pouco, esses itens sejam incluídos nas iterações.

Poderíamos ter itens como "Refatorar Método de 2000 linhas na Classe de Transferência Bancária", ou "Atualizar Biblioteca POI para Versão mais Recente para evitar

Vazamento de Memória", ou "Aumentar Cobertura de Testes do Módulo de Empréstimos" etc.

Alguns tipos de dívida técnica podem ser deliberadamente deixados de lado, por não haver valor em resolvê-los. Por exemplo, em casos de produtos próximos ao término de seu clico de vida, ou de um protótipo com objetivo único de aprendizagem, ou produtos com ciclos de vida muito curtos (como projetos para campanhas de marketing pontuais, por exemplo). É importante que o time foque naquilo que de fato trará benefícios para a produtividade do time e qualidade do produto.

E então? Seu time está pronto para pagar suas dívidas? Lembre-se que assim como nas dívidas financeiras, quanto mais tempo levamos para pagar, mais difícil, porque paga-se com juros.

Nada de Janelas Quebradas

Pesquisas no campo de criminalidade e decadência urbana, descobriram que uma janela quebrada pode rapidamente levar um prédio limpo e intacto a se tornar um prédio decadente e abandonado [68].

A Teoria da Janela Quebrada afirma que uma janela quebrada, se não consertada por algum tempo, transmite para as pessoas um senso que ninguém se importa com o prédio [61]. Então, de repente, outra janela é quebrada, lixo se acumula, paredes são pichadas, em pouco tempo o prédio fica tão prejudicado que o dono já não consegue mais repará-lo, então o que antes era apenas um senso de abandono, agora se torna realidade.

Da mesma forma, quando um desenvolvedor deixa um código sem cobertura testes, por exemplo, o próximo desenvolvedor tem essa sensação que não houve zelo por aquele código, e cria um novo método, deixando-o sem testar também.

Com o tempo será possível notar uma grande queda na cobertura de testes, tanto que talvez se torne muito difícil para a equipe reverter o cenário. O mesmo poderia valer para código deixado sem refatorar, ou qualquer outra má prática de desenvolvimento.

Quando uma janela é deixada quebrada, outra será quebrada, e outra, e outra... Por isso, não deixe janelas quebrada se achar uma, e nunca seja o primeiro a quebrar uma janela.

4.12 Agilidade Explícita com Mural de Práticas

"Nós somos aquilo que fazemos com frequência. Excelência, então, não é um ato, mas um hábito"
– Aristóteles

Desculpe-me, mas eu tenho que dizer: "Não é o quadro na parede que faz da sua equipe uma equipe ágil".

O filósofo grego Aristóteles que viveu há mais de 300 A.C. já reconhecia que nós somos frutos dos nossos hábitos, das nossas ações frequentes e constantes. Nessa linha de pensamento, você não pode dizer que é uma pessoa caridosa, só porque uma vez na vida fez uma doação para uma causa beneficente, não pode dizer que é uma pessoa aventureira se quando tinha 17 anos de idade pulou de bungee jump no parque de diversões.

Você também não pode dizer que sua equipe é ágil só porque 1 dos 25 desenvolvedores do seu time, uma vez, tentou aplicar desenvolvimento guiado por testes em seu tempo livre, e nem pode dizer que sua organização tem uma cultura de aprendizagem só porque há um espaço para palestras uma vez no ano. De jeito nenhum. Aristóteles sabia bem do que estava falando.

O ponto é que "nós podemos dizer que somos ágeis porque nós estamos frequentemente utilizando práticas ágeis" e "nós podemos dizer que nossa organização tem uma cultura de aprendizagem porque estamos aprendendo algo novo todos os dias".

Quanto mais eu treino, mais sorte eu tenho.
– Tiger Woods

A pergunta é: quais são as práticas que fazem de nós uma equipe ágil? E uma ferramenta muito simples para ajudar as equipes a se fazerem essa pergunta com frequência é o *mural de práticas*, que consiste basicamente em tornar explícitas todas as práticas de uma equipe através de pequenos ícones expostos em uma área do quadro da equipe (veja na figura 4.5.

Figura 4.5: Mural de Práticas

Inclua todas as práticas do seu time, aquelas das que vocês se orgulham e das que não se orgulham, torne tudo explícito para que possam visualizar e discutir frequentemente se elas ainda fazem sentido e quando novas práticas poderão ser acrescentadas.

Você pode inclusive associar métricas a cada uma dessas práticas para assegurar que elas realmente fazem parte do dia a dia do time e não estão no mural apenas de enfeite.

Lembre-se, nós somos aquilo que fazemos com frequência. O que você e seu time estão fazendo com frequência, que os tornam ágeis?

4.13 E agora, o que eu faço amanhã?

Seu time já possui uma definição de pronto para histórias, iterações e *releases* explícita e visível para todos. Ainda não? Que tal reunir o time e o PO para propor e discutir uma definição? Se vocês já possuem definições de pronto, faça uma revisão e verifique se ela está atualizada e se está, de fato, representando o estado de pronto ideal para o contexto da sua equipe.

Reúna o time e identifique as práticas ágeis que fazem parte do dia a dia do time, construa um mural de práticas e torne tudo explícito. Na próxima retrospectiva,

discuta com o time as práticas atuais, se todas elas ainda fazem sentido, e que novas práticas poderiam ser acrescentadas para benefício do projeto e da equipe.

Reconheça e Pague suas Dívidas Técnicas: Reúna sua equipe, e faça um levantamento da dívida técnica do seu projeto atual. Classes difíceis de se compreender ou alterar, código sem cobertura de testes, gambiarras deixadas para trás etc. Levante tudo que puder, e procure fazer priorizar junto com o time de acordo com o quanto cada um dos itens prejudica a performance do time. Leve a lista ao PO, e negocie com ele de que forma esses itens podem ser solucionados ao longos das iterações.

Sua equipe tem uma política para programação em par? Tudo é feito em par? Nada é feito par? Define-se o que será ou não feito em par na Reunião Diária? Que tal conversar com equipe sobre isso e definir alguma coisa para a próxima iteração?

Que tal propor uma sessão de revisão de código com alguns membros do seu time? Durante a sessão apresente a seu time alguns dos conceitos de Clean Code vistos nesse capítulo.

Quais métodos vocês estão utilizando para guiar o processo de desenvolvimento? Será que podem incluir algumas práticas de outros métodos para aprimorar o processo?

Capítulo 5

Otimizando Valor

Esse é o terceiro nível de fluência no qual a equipe passa a entregar ainda mais valor agregado e torna-se capaz de tomar melhores decisões para os produtos em desenvolvimento.

Nesse momento a equipe compreende muito bem o que o mercado está pedindo, assim como quais são as necessidades de negócio, e como satisfazê-las.

Agora, a equipe, apresenta seus resultados através de métricas concretas de negócio como ROI (Retorno sobre o Investimento), Margem de Lucro Líquido por Colaborador, Satisfação do Cliente etc [57].

Larsey e Shore afirmam que para se atingir esse nível de fluência a equipe precisará de membros especializados em negócio trabalhando em tempo integral como parte da equipe (analistas de negócio, gerentes de produto, vendedores, publicitários, entre outros de acordo com o domínio de negócio do produto em desenvolvimento).

5.1 Direcionando a Equipe

"Para quem não sabe para onde vai, qualquer caminho serve."

– Lewis Carroll, Alice no Pais das Maravilhas

Definir a visão, os propósitos, objetivos e metas da organização e do projeto em que o time está trabalhando é uma ótima forma de mostrar a todos a direção para qual devem juntos levar a organização através do trabalho realizado no dia a dia. Tudo isso são ferramentas para unir e direcionar as pessoas com algumas mudanças sutis em termos de abrangência e foco. Nós vamos ignorar essas sutilezas e nos referir apenas a **metas**.

Uma diferença crucial entres metas tradicionais e metas ágeis é que as características e critérios para definição das ágeis não são fixas, mas variáveis de acordo com o contexto [8].

Dependendo do objetivo final, algumas metas podem ser mais inspiradoras, ou memoráveis, ou específicas, ou ambiciosas, enfim, as características da meta não precisam atender a um conjunto de critérios como SMART (Específicas, Mensuráveis, Atingíveis, Realistas, Tangíveis) por exemplo, para que sejam úteis para guiar o time.

A Wikimedia, fundação por trás da famosa enciclopédia digital Wikipedia, apresenta sua missão como: "A missão da Fundação Wikimedia é empoderar e engajar pessoas pelo mundo para coletar e desenvolver conteúdo educacional sob uma licença livre ou no domínio público, e para disseminá-lo efetivamente e globalmente." [26], já sua visão é "Imagine um mundo onde cada ser humano possa compartilhar livremente na soma de todo o conhecimento. Esse é o nosso compromisso."[27].

Note que ambas de uma forma ou de outra mostram exatamente o que a Wikimedia está fazendo, e no que seus colabores devem concentrar seus esforços para conquistar. Esse é o ponto.

Jamais use metas para intimidar as pessoas ou ameaçá-las, em vez disso, para ajudá-las a entender o que a organização precisa que façam naquele dado momento, e para que todos possam "remar o barco na mesma direção".

Não associe metas a recompensas financeiras, pesquisas apontam que isso pode destruir a colaboração e tirar o foco das pessoas do que realmente importa [47], é importante que a realização da meta seja a própria recompensa para o time.

É importante que as metas sejam comunicadas e permaneçam sempre explícitas e visíveis para o time. De que vale uma meta se as pessoas que precisam realizar o trabalho para que elas sejam atingidas não a conheçam?

Metas são ferramentas, e como qualquer outra ferramenta, podem ser bem ou mal utilizadas. Quando bem usadas podem unir e direcionar o time, quando mal usadas podem dividir as pessoas, quebrar a colaboração e desmotivar.

Faça bom uso de metas!

5.2 Métricas Ágeis

"Tudo que pode ser contado não necessariamente conta; Tudo que conta não necessariamente pode ser contado."
– Albert Einstein

Métricas podem ser úteis para ajudar o time a compreender onde estão, e ajudá-los a comparar com o estado em que querem estar. Assim como as metas, as métricas também devem estar sempre explícitas e visíveis para o time.

O Scrum, por exemplo, sugere que as equipes utilizem burndown charts para que todos possam ter feedback de como está o andamento da iteração (estado atual) em relação ao planejamento (estado desejado).

Na figura 5.1 podemos ver um exemplo de burndown, em que a equipe não conseguiu entregar tudo o que foi planejado. O eixo vertical representa o total de trabalho pendente, nesse caso pontos de história de usuário (poderia ser qualquer outra medida de trabalho de pendente), enquanto o eixo horizontal representa o tempo, nesse caso dias da semana (poderia ser qualquer outra medida de tempo como iterações ou semanas, por exemplo).
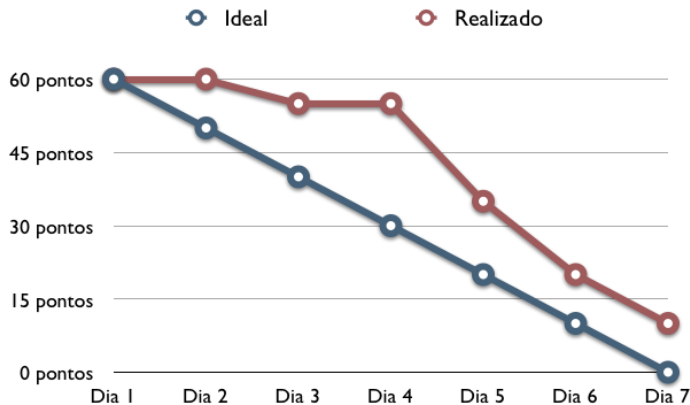
Figura 5.1: Burndown Chart

Note que no dia 4 há uma diferença grande entre o ideal e o realizado e, é uma boa oportunidade para que o time possa discutir sobre o que está acontecendo e tomar alguma ação para eventualmente melhorar e investir no sucesso da iteração, enquanto ainda há tempo.

No burndown chart, sempre que algum trabalho for concluído, o gráfico cai, quando o trabalho é reestimado, o gráfico sobe, quando algum trabalho é acrescentado, o gráfico sobe, e quando algum trabalho é removido, o gráfico desce [19].

Uma alternativa ao burndown chart, é burnup chart, que em vez de descer à medida que o trabalho vai sendo concluído, vai subindo. Alguns times preferem usar o burnup porque é mais fácil de expressar alterações do escopo, como por exemplo, uma melhoria ou defeito urgente que surgiu ao longo de uma iteração, como pode ser visto na Figura 5.2.
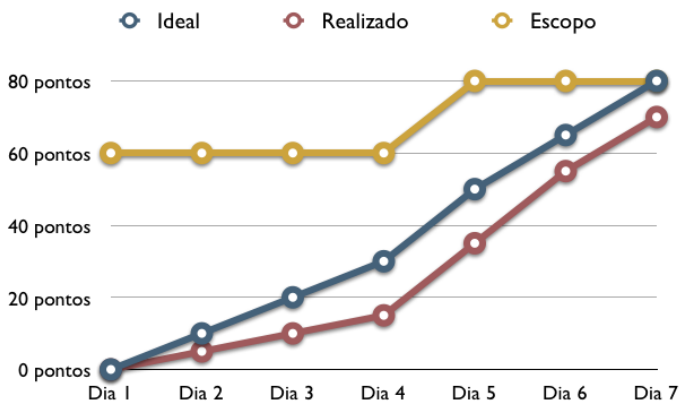
Figura 5.2: Burnup Chart

Equipes que aplicam o método Kanban, geralmente, utilizam um gráfico semelhante ao burnup chart, porém detalhado por etapa do processo, permitindo visualizar a variação do trabalho em progresso ao longo do tempo, trata-se do diagrama de fluxo cumulativo (em inglês Cumulative Flow Diagram ou CFD), veja na figura 5.3.
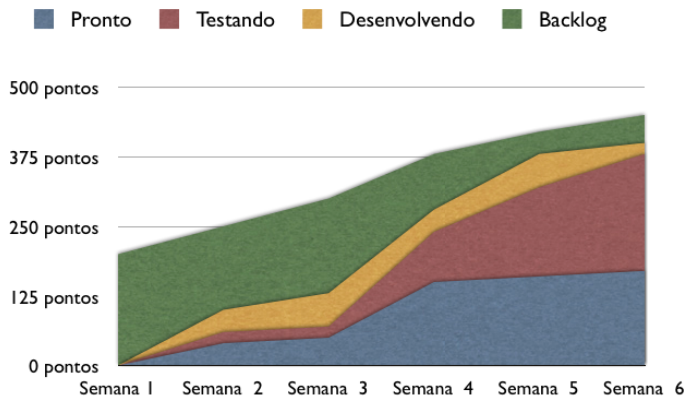
Figura 5.3: Cumulative Flow Diagram

Com o CFD, é possível observar exatamente onde o trabalho em progresso está, e quais os gargalos. Observe na figura 5.3, que depois da semana 3 a etapa "testando" não parou de aumentar de tamanho, diminuindo as entregas (etapa pronto). Isso pode ser um sinal que os testes se tornaram um gargalo e que algo precisa ser feito para tornar o processo mais eficiente, aumentando o *throughput* (entregas).

Na grande maioria das vezes, esse acumulo de trabalho em determinada etapa do processo é um sinal de que é necessário trabalhar os limites de trabalho em progresso conforme estudamos na seção 3.5.

Depois de algumas iterações é possível medir também a velocidade do time por iteração (total de pontos entregues por iteração) e a velocidade média, veja na figura 5.4.
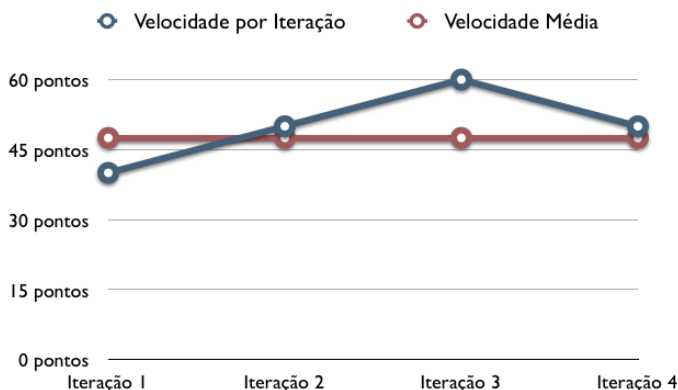
Figura 5.4: Gráfico de Velocidade

A velocidade, o escopo pendente/entregue e o trabalho em progresso são as métricas mais acompanhadas por equipes ágeis, mas não são as únicas. Por isso, agora estudaremos os tipos de métricas, e quais outras podem trazer visibilidade e **feedback** para a equipe e os interessados no projeto.

Tipos de Métricas

Jurgen Appelo e Mike Cohn recomendam que se use dois tipos de indicadores [8][20]: Indicadores Indutores e Indicadores de Resultado.

Indicadores Indutores (*leading indicators*) ajudam a detectar um aumento na probabilidade ou na gravidade de um evento antes que apareçam nos indicadores de resultado.

Quando um indicador indutor muda, significa que você pode estar no caminho para atingir um objetivo ou meta. Por exemplo, aumentar a cobertura de testes ou a quantidade de testes criados pode indicar mais qualidade no produto.

Você pode ter diversos objetivos em uma equipe de desenvolvimento de software, como por exemplo, aumento na qualidade, aumento na produtividade, produção de melhores estimativas, redução do custo do desenvolvimento, maior previsibilidade do cronograma, aumento na satisfação do usuário, releases mais frequentes etc. Para cada objetivo você deve ter ao menos um indicador indutor e um de resultado.

Já os Indicadores de Resultado (*lagging indicators*) ajudam a confirmar que um evento pode ocorrer e reagem mais lentamente às mudanças no ambiente do que

os indicadores indutores. São métricas que verificam que você atingiu seu objetivo depois de ter completado o trabalho. Por exemplo, reduzir o número de defeitos reportados por clientes aponta que a qualidade do produto de fato melhorou.

Meça times e não indivíduos

Quando as pessoas acreditam que serão afetadas pelos resultados de métricas sobre seus trabalhos individuais, elas rapidamente encontram formas de corromper tais métricas [38], além disso, medir indivíduos pode destruir a colaboração do time, uma vez que ajudar um outro membro do time pode significar melhorar a métrica de desempenho dele e prejudicar a sua própria.

Já quando mede-se o time em vez do indivíduo, já que a colaboração tende a melhorar a performance do time como um todo, a colaboração entre as pessoas será mais incentivada.

Esses são alguns dos indicadores mais comum em ambientes ágeis:

- Velocidade do Time: Quantidade de pontos de história que um time consegue entregar em uma determinada iteração.
- Aceleração: Alteração da velocidade do time, ao longo das iterações.
- Lead time: O tempo entre o pedido do cliente e a entrega (o início e o fim do processo). Ex: 2 dias do backlog (registro do pedido do cliente) à produção (entrega do software funcionando).
- Cycle time (tempo de ciclo): O tempo total entre o início e o fim da produção ou da prestação de serviço. Ex: 6 horas por história de usuário do "to do" ao "done".
- Tempo de Vida: Quantidade de tempo em que uma história de usuário está no Backlog.
- Quantidade de histórias impedidas: Lista de histórias impedidas por falta de informações ou recursos.
- Taxa de Defeitos por história: A quantidade defeitos para cada história de usuário.

- Saúde do Build: Tempo que o Build está passando em relação ao tempo que está quebrado.
- Valor de Negócio Agregado: Total de valor de negócio das histórias entregues na iteração.
- Horas Trabalhadas: Soma das horas dedicadas ao projeto por todos os membros do time.
- Cobertura de Testes: Percentual de cobertura de testes de unidade. Pode ser facilmente obtido por ferramentas de análise de cobertura.
- Quantidade de Cenários de Teste: Contagem dos cenários de testes disponíveis.
- **Throughput** (Produção): Quantidade de histórias entregues em determinado período de tempo.
- Happiness Index da Equipe: Indica a Motivação do Time, é subjetivo, mas pode ser muito útil para perceber quando o clima está ruim e tomar alguma ação para melhorar.
- Trabalho em Progresso (WIP): Quantidade de histórias começadas e não entregues.
- Lucro: Influencia das entregas no lucro obtido através dos incrementos no produto em desenvolvimento.

Esses são apenas alguns indicadores que podem ou não fazer sentido no seu contexto. As possibilidades são infinitas, porém, procure encontrar as métricas que mais podem agregar valor ao time, e dê visibilidade a elas para que o time possa acompanhá-las e melhorar a cada nova iteração.

Mas quem é Responsável pela Métricas?

No método XP há um papel importante, que muitas vezes é esquecido, trata-se do **tracker** . O **tracker** acompanha a agenda do time, e algumas métricas, sendo a mais importante delas a velocidade do time, que é a relação de tempo ideal estimado para as tarefas e o tempo gasto na implementação.

Ele deve manter seus olhos nos dados importantes a serem acompanhados, como a variação da velocidade, horas extras trabalhadas, a relação de testes que passaram

e que falharam, a quantidade de defeitos, e métricas como **lead time** e o tempo de ciclo, além de métricas mais técnicas voltadas ao código fonte como cobertura de testes, métricas de complexidade, coesão e acoplamento.

Martin Fowler e Kent Beck resumem que o papel do **tracker** diz respeito a fazer perguntas simples que apontem possíveis problemas [28].

5.3 Apresente o Resultado em Reuniões de Demonstração

A cada novo incremento que é realizado no produto, é interessante reunir todos os envolvidos no projeto para uma reunião de demonstração.

Nessa reunião a equipe apresenta o que há de novo no produto demonstrando as novas funcionalidades. No Scrum, essa reunião é chama "Sprint Review Meeting".

Nas Reuniões de Demonstração apresentações de Power Point não são recomendadas [18], em vez disso, apresenta-se o software em funcionamento. Funcionalidades que não estão prontas não devem ser demonstradas.

Durante a reunião, os interessados no produto podem fazer comentários, observações e manifestar desejos de mudanças nas funcionalidades apresentadas, tudo isso pode ser anotado e tratado posteriormente pelo *Product Owner*.

Para uma iteração de 1 mês, Ken Schwaber recomenda um tempo máximo de 1 hora de preparação e 4 horas de duração [55].

O *Product Owner* não precisa necessariamente aguardar até a reunião de demonstração para ver o incremento a ser feito no software e dar feedback a equipe. Algumas equipes tem a prática de "**Just-in-Time Reviews**" [46], isso é, assim que a história ficar pronta, a equipe se reune com o *Product Owner* para uma rápida demonstração.

5.4 Melhoria Contínua com Retrospectivas

Cada equipe é única e possui características diferentes, nenhum processo é perfeito e, raramente, cobrirá todas as necessidades de uma equipe ou projeto. Por essa razão, é necessário trabalhar continuamente na melhoria e adaptação do processo. As reuniões de retrospectiva são uma excelente forma para fazê-lo.

As retrospectivas são naturais, afinal todo final de ano, você, provavelmente, faz a sua lista de desejos para o ano novo e uma reflexão sobre tudo que conquistou no ano que está acabando. De forma semelhante, ao final de cada iteração as equipes

se reúnem para uma retrospectiva que visa levantar todos os acontecimentos importantes da iteração e refletir sobre o que foi bom, o que deve ser mantido e o que pode ser melhorado.

Nessas reuniões todos os membros da equipe têm a oportunidade de falar sobre o que está ou não funcionando bem, sugerir novas práticas, tecnologias, e apresentar outras ideias em geral, dessa forma todos podem contribuir para a melhoria do processo.

Outro ponto muito importante das retrospectivas é o levantamento dos acontecimentos mais significantes da iteração, quando a equipe apresenta as principais lições aprendidas que serão levadas para o futuro.

As retrospectivas não têm como foco apenas questões voltadas ao lado técnico do desenvolvimento de software, mas vão além, e devem tratar também do lado humano e do relacionamento entre as pessoas envolvidas no projeto. Isso, segundo Esther Derby e Diana Larser, duas das maiores especialistas no assunto, são questões tão desafiadoras quanto ou mais desafiadoras do que as questões técnicas [22].

Nenhum processo é perfeito! Além disso, sua equipe é única, e provavelmente está sempre enfrentando desafios diferentes de todas as outras, por isso, o processo deve ser adaptado e melhorado continuamente para atender de forma eficiente as necessidades da equipe. É exatamente nisso que as retrospectivas podem ajudar um equipe de desenvolvimento de software.

Nas retrospectivas é possível que a equipe reflita e identifique mudanças e melhorias que poderão aumentar a qualidade do software e aprimorar seu dia a dia de trabalho.

Retrospectivas são naturais em métodos ágeis que focam em adaptação e respostas rápidas a mudanças, porém, podem também ser realizadas em equipes que utilizam métodos tradicionais.

James Shore e Shane Warden, sugerem que apenas os membros da equipe participem de retrospectivas [66], para que esses possam sentir-se à vontade e que efetivamente possam falar abertamente de seus problemas do cotidiano. Deve-se entender por "equipe" todas as pessoas envolvidas no projeto que são diretamente ligadas ao desenvolvimento do software, isso exclui, por exemplo, clientes, investidores e curiosos.

Não Subestime o Poder da Retrospectiva

Retrospectivas, infelizmente, costumam ser a primeira prática a ser cortada do processo em momentos que a equipe está sob pressão, mas ocorre que, geralmente, a retrospectiva é a melhor oportunidade que o time tem de entender porque está nessa situação de pressão e o que pode fazer para sair dela.

O Facilitador de Retrospectivas

Toda reunião de retrospectiva deve ter um facilitador (também chamado de líder de retrospectiva). O facilitador deve garantir que todos os participantes mantenham o foco nos objetivos da retrospectiva e ajudar a equipe a se organizar para realizar as atividades que os direcionem a descobrir o que precisa ser melhorado e o que pode ser feito para melhorar.

Os facilitadores focam na estrutura e no processo da retrospectiva, e devem estar constantemente atentos às atividades e ao tempo. É importante que o tempo de duração da reunião seja bem definido, apresentado a todos, assim como respeitado. Ultrapassar o tempo definido pode fazer com que os participantes fiquem dispersos e sintam-se desrespeitados, uma vez que podem ter outros compromissos, atividades e obrigações que requerem sua atenção após a reunião. Reuniões que não terminam no tempo definido tendem a se tornar cansativas e pouco produtivas.

Os outros participantes da retrospectiva, em contrapartida, devem estar totalmente focados no conteúdo, na discussão e na tomada de decisões, completamente despreocupados com o processo da retrospectiva, deixando-o a cargo do facilitador.

O papel de facilitador pode ser rotativo dentre os membros da equipe, de forma que a cada iteração possa existir uma pessoa diferente assumindo esse papel. De fato, é recomendável que cada reunião de retrospectiva possua um facilitador diferente, de modo que todos possam exercitar esse papel e treinar as habilidades por ele exigidas.

O facilitador deve permanecer neutro em discussões mesmo quando houver fortes opiniões a respeito do tema em discussão. Se ele estiver engajado na discussão, provavelmente não conseguirá fazer um bom trabalho na liderança da retrospectiva.

O facilitador deve evitar que ocorram conversas em paralelo, e deve assegurar que todos sejam devidamente ouvidos. Contudo, ele deve ainda cortar as discussões quando estiverem propensas a estourar o tempo da reunião ou impedir que todos os

tópicos importantes sejam abordados.

É essencial que todos tenham oportunidade de falar e que os mais faladores não dominem a reunião. O facilitador deve estar sempre atento às pessoas que falam demais ou de menos e incentivar pessoas com maior dificuldade de expressar seus pensamentos em grupo a falarem.

Outra responsabilidade do facilitador da retrospectiva é lidar com comportamentos inadequados ao longo da reunião. Larsen e Debby aconselham evitar chamar a atenção das pessoas individualmente, buscando sempre chamar atenção do grupo enquanto for possível, enfatizando o comportamento e não a pessoa.

Além disso, deve-se evitar que as pessoas utilizem a retrospectiva para apontar culpados, transformando-a em algo conhecido como jogo de culpas, o que geralmente faz com que as pessoas "levem para o lado pessoal", percam o foco da retrospectiva e criem um clima desagradável e competitivo, impedindo que o grupo trabalhe colaborativamente.

Ao invés de culpar alguém por ter quebrado o build, por exemplo, deve-se procurar expor problemas por falta de atenção ao enviar o código ao repositório sem antes executar os testes de unidade, e ressaltar a importância de manter-se alerta a essa prática. Foco está sempre no problema e não na pessoa (ou possíveis culpados).

Norm Kerth em seu livro "Project Retrospectives" apontou uma frase, a qual chamou de primeira diretriz das retrospectivas:

"Independentemente do que descobrirmos hoje, nós compreendemos e acreditamos que todos fizeram o melhor trabalho que podiam, dado o que conheciam, suas competências e habilidades, os recursos disponíveis, bem como a situação em mãos. [32]"

Kerth recomenda que no início das retrospectivas o facilitador leia a primeira diretriz para a equipe. Ela é uma ferramenta para trazer consciência para o trabalho colaborativo e para a confiança que os membros da equipe devem depositar entre si.

Dessa forma, é possível evitar problemas como ataque a indivíduos e distribuição de culpa. É essencial que as pessoas deixem de lado a competição e colaborem para alcançar um bom resultado na retrospectiva, podendo estender isso também ao trabalho do dia a dia.

Aprenda sobre a história e o ambiente da sua equipe. É importante analisar a história e moral da equipe, especialmente em relação ao projeto atual.

Estude quais são os artefatos disponíveis e quais podem ser incorporados para ajudar, converse com líderes formais e informais. Essa investigação poderá ajudá-lo

a fazer as perguntas certas durante a reunião e o ajudarão a prever quais problemas devem ser enfrentados.

Em suma, o facilitador deve guiar a reunião, assegurar que todos os membros da equipe estejam compreendendo os objetivos das atividades e que todos tenham chance de falar e expor seus pensamentos.

Quanto tempo deve levar uma reunião de retrospectiva?

Não existe uma fórmula para definir este valor. Como costumam dizer os consultores: depende. Você deve levar em consideração alguns fatores como o tempo da iteração, o tamanho da equipe, propensão a conflitos e controvérsia, e complexidade dos assuntos a serem discutidos. O método Scrum, por exemplo, sugere duas horas de reunião de retrospectiva para um sprint de 30 dias.

A facilitação e até mesmo a participação em retrospectivas é um processo de aprendizado contínuo: quanto mais prática, maior será a eficiência para alcançar bons resultados. Você deve avaliar o resultado das retrospectivas através dos benefícios conquistados, isto é, das melhorias realizadas por meio das ações que foram definidas e efetivamente realizadas.

As 5 etapas das retrospectivas

Diana e Derby propõem cinco passos básicos para a realização eficiente de uma reunião de retrospectiva ágil [22]. Esses passos ajudarão o facilitador a guiar a reunião e levar os membros da equipe a trazer à tona possíveis itens a serem melhorados e ideias para alcançar as melhorias.

1) Preparação

2) Apresentação dos Dados

3) Insights

4) Decidir o que fazer

5) Fechamento

Abrindo a Retrospectiva

Antes de tudo é importante preparar as pessoas para reunião para que foquem no trabalho que deve ser realizado ao longo da retrospectiva. Crie a atmosfera ideal para que as pessoas sintam-se confortáveis em discutir tópicos complexos e estabelecer diálogos desafiadores.

Defina uma meta para a retrospectiva. Uma meta bem clara oferece às pessoas um senso de razão pela qual estão investindo seu tempo. Procure escolher uma meta abrangente para que a equipe fique livre para utilizar sua criatividade e alcançar insights.

Considere o contexto da equipe e descubra metas que a ajudarão. É certo que toda equipe tem suas particularidades, diferentes pontos fortes e fracos. No momento de elaborar a meta da retrospectiva, procure algo que represente a busca das melhorias que podem trazer maior beneficio para sua equipe em particular.

Apresentando Dados

Depois da preparação, vem a apresentação dos dados. Ainda que os benefícios de reunir informações de uma iteração que tenha durado uma ou poucas semanas possa não ser evidente, essa é uma prática muito importante para a retrospectiva.

Em uma iteração semanal, por exemplo, perder um dia significa perder 20% dos acontecimentos. Mesmo que as pessoas estejam presentes, elas não veem tudo o que acontece. Além disso, pessoas diferentes veem os mesmos acontecimentos com perspectivas diferentes. Portanto, reunir informações faz com que a equipe crie uma visão compartilhada de tudo o que aconteceu, expandindo a perspectiva de toda ela.

É importante apresentar métricas, histórias de usuário concluídas, decisões tomadas, resultados de outras reuniões realizadas ao longo da iteração, desafios enfrentados, novas tecnologias adotadas e tudo que tiver significado para a equipe.

Você pode utilizar como métricas o burndown chart, a velocidade da equipe, o número de histórias prontas, o número de defeitos resolvidos, a cobertura de testes etc.

Outra técnica interessante para apresentar os acontecimentos da iteração é criar uma linha do tempo e permitir que as pessoas enumerem acontecimentos. Isso faz com que as pessoas lembrem-se dos eventos relevantes ao projeto, como releases, milestones e problemas diversos que podem ser posteriormente analisados.

Independente da técnica utilizada, a equipe deve descrever os acontecimentos e principalmente os problemas que estão enfrentando, isto é, devem refletir e falar

sobre o que não está indo bem. Abaixo apresentamos alguns exemplos de problemas comuns que são levantados por equipes nessa etapa da retrospectiva:

- Não estamos fazendo testes de unidade;
- Temos dúvidas sobre as histórias de usuário e ninguém está disponível para nos ajudar.
- Há muito retrabalho.
- Falta de conhecimento técnico.
- Muitos defeitos.

Gerando Insights

Depois de analisar os dados é hora de perguntar os porquês, pensar sobre o que se pode mudar e identificar o que fazer de forma diferente.

A equipe deve utilizar os dados reunidos para identificar os pontos fortes e fracos da última iteração, e então expor suas ideias para fortalecer os pontos fracos e não permitir que os pontos fortes enfraqueçam.

Insights são grandes ideias que a equipe acredita que, se aplicadas, serão eficientes na obtenção de melhores resultados. Busque por condições, interações e padrões que contribuíram para o sucesso em melhorias anteriores para encontrar novos insights.

Esses insights ajudarão a equipe a descobrir formas de trabalhar com maior eficiência, que é o grande objetivo da retrospectiva. Ao identificar problemas, evite simplesmente aceitar a primeira solução que for proposta. Ela pode de fato ser a mais adequada, porém, vale considerar outras possibilidades, analisar causas e efeitos e permitir que a equipe colaborativamente alcance a melhor solução.

Brainstorming é uma técnica interessante para instigar a equipe a ter insights. A técnica consiste em dar à equipe um tempo para pensar, expor e construir ideias sobre outras ideias anteriormente propostas.

Depois de levantado um número suficiente de propostas para resolver problemas ou alcançar melhorias, a equipe deve definir filtros e aplicá-los para reduzir o número de ideias e então levá-las à próxima etapa para efetivamente decidir o que fazer.

Uma técnica interessante para resolver isso, é a **votação por pontos** (figura 5.5. Imagine que há 20 proposta. Escreva cada uma em um cartão e dê a cada membro

da equipe 5 pontos. Então, cada membro pode investir seus pontos em nos cartões. Tanto faz se quiser investir os 5 pontos em um único cartão ou 1 ponto em 5 cartões diferentes. Os cartões com maior número de pontos serão os escolhidos.
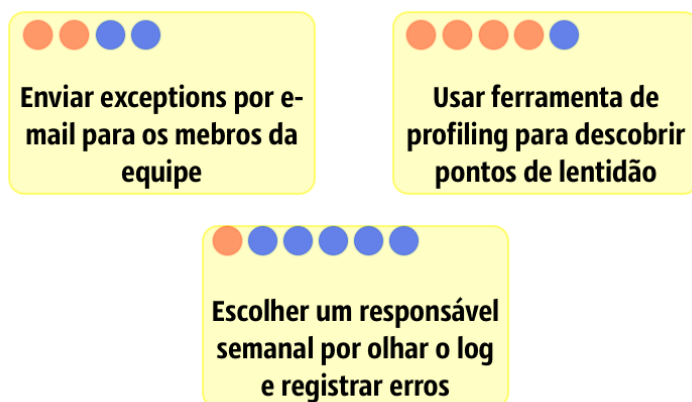
Figura 5.5: Votação por Pontos

Outra ferramenta poderosa para obtenção de insights é a técnica dos 5 porquês. Com dados e fatos em mãos, a equipe pode realizar essa atividade para melhor compreender as razões pelas quais determinados eventos ocorreram.

A técnica consiste em analisar um determinado acontecimento perguntando por que ele aconteceu, e para a resposta a essa pergunta questiona-se novamente o seu porquê, e assim sucessivamente até alcançar cinco respostas (veja um exemplo na figura 5.6.
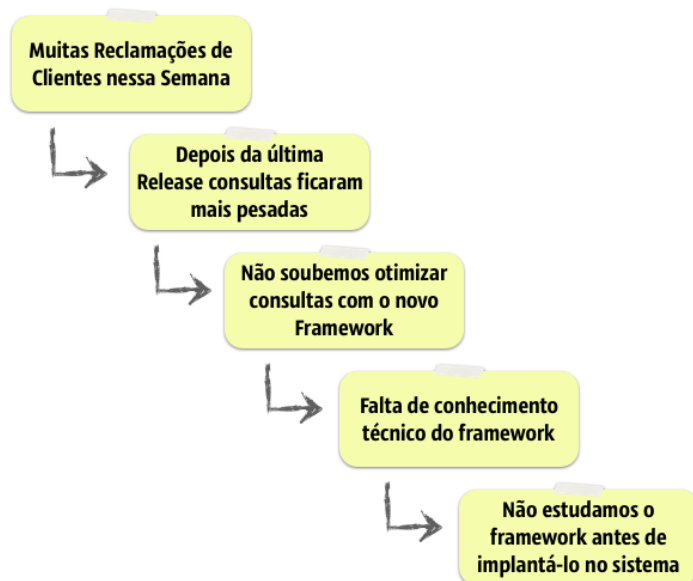
Figura 5.6: Os 5 porquês

Essa técnica, muito utilizada para resolução de problemas, foi desenvolvida por Sakichi Toyoda na Toyota e faz parte de métodos como Lean Manufacturing e Six Sigma. Identificando a causa-raiz do problema, deve-se então trabalhar em sua solução, ou seja, buscar insights.

Esse exercício pode ser realizado por toda a equipe em um grupo grande, ou em pares, que depois apresentam o resultado a todo o grupo para consolidação das ideias.

Definindo as Ações da Retrospectiva

O quarto passo tem o objetivo de ajudar a equipe a decidir o que deve ser feito. Agora que a equipe tem uma lista de potenciais ideias de experiências e melhorias a serem realizadas, é preciso selecionar os itens considerados mais importantes (geralmente um ou dois por iteração) e planejar o que fazer, ou seja, que ações tomar. É neste momento que o foco da reunião é mudado do passado para o futuro, isto é, da iteração anterior para a próxima iteração.

É importante que a equipe esteja comprometida com os itens que foram selecionados para serem melhorados na iteração. Um exemplo de ação a ser tomada em

uma nova iteração é: "Todos os membros da equipe trabalharão em par dois dias por semana".

Evite a todo custo sair da retrospectiva sem uma lista clara de ações a serem tomadas para alcançar melhorias. Mantenha sempre o foco em coisas que efetivamente podem ser melhoradas e evite investir muito tempo em questões que não dependem da equipe ou não podem ser mudadas, como mau tempo, trânsito etc.

É importante que a responsabilidade de realizar as ações definidas na retrospectiva seja compartilhada e dividida por toda a equipe. Não permita que apenas um ou poucos membros da equipe constantemente tomem essa responsabilidade para si impedindo que outros colaborem.

Encerrando a Retrospectiva

Finalmente, no quinto passo, a equipe deve definir como documentar o que foi decidido e aprendido na reunião. Para isso, algumas equipes fotografam o que foi escrito na lousa, outras elegem um membro da equipe para criar uma pauta, outros escrevem todos os itens em cartões e os guardam, isso fica a critério da equipe.

É importante que as decisões tomadas fiquem disponíveis de alguma forma para consulta de todos e utilização em retrospectivas futuras. Algumas equipes mantêm todas as melhorias definidas na reunião de retrospectiva em um **backlog** de melhorias, que é mantido e priorizado pelo próprio time.

Comece com o formato mais Simples

Tanta informação, etapas, tempo, preparação, técnicas etc. podem parecer um pouco demais para quem está começando. Não se intimide com isso, comece reunindo o time em frente a um quadro branco ou um flip-chart e discuta com o time sobre as seguintes perguntas:

- O que está indo bem?
- O que pode ser melhorado?

No final das contas o que realmente importa é que a reunião tenha como resultado ações a serem tomadas pela equipe para que a melhoria contínua seja aplicada, e que na próxima retrospectiva, a equipe seja melhor do que era na última.

A medida que o time for evoluindo na utilização de métodos ágeis, procure variar um pouco as técnicas utilizadas para que a reunião não se torne previsível e massante. Dinamizar e alterar o formato da retrospectiva sem dúvida estimulará a criatividade do time para buscar soluções para melhorar continuamente.

Você pode pesquisar por novas técnicas e dinâmicas de grupo para utilizar nas retrospectivas nos sites: http://retrospectivewiki.org e http://tastycupcakes.org.

5.5 Eliminando Desperdícios com Lean

Taiichi Ohno, criador do Toyota Production System (TPS), definiu-o como um "sistema de gerenciamento para a eliminação absoluta de desperdícios", e disse que o que deve ser feito é uma linha do tempo entre o momento que o cliente faz um pedido até o momento em que recebe o produto e paga por ele, ou seja o processo completo.

Depois de identificar todas as etapas dessa linha do tempo, deve-se reduzi-la, removendo tudo o que não agrega valor, ou seja, deve-se eliminar todo o desperdício. Em suma, qualquer atividade que consuma tempo ou dinheiro e não agrega valor é desperdício.

Desperdício com Trabalho Parcialmente Pronto

Para indústrias em geral, acumular estoque é considerado desperdício. O estoque precisa ser movimentado de um lado para o outro, armazenado, rastreado, pode ser perdido, roubado, ficar obsoleto, quebrar etc. É por isso que o ideal é ter o mínimo de estoque possível.

Você deve estar se perguntando: qual é a relação de estoque com software? Estoque na perspectiva de software, para Poppendieck, representa trabalho parcialmente pronto, trabalho que pode ficar obsoleto, pode ser perdido [48].

Trabalho parcialmente pronto pode ser, por exemplo, requisitos especificados com excesso de detalhes antes do desenvolvimento ser iniciado. Lembre-se que requisitos podem mudar e tornar-se obsoletos, ou ainda histórias de usuários desenvolvidas aguardando para serem validadas por uma equipe de qualidade, ou aguardando para passar por um processo de integração. Tudo isso é trabalho parcialmente pronto e consequentemente desperdício.

Desperdício com funcionalidades que não agregam valor

A maior fonte de desperdício pode ser vista claramente no Chaos Report [3]: somente 20% das funcionalidades desenvolvidas em software são efetivamente utilizadas regularmente; os outros 80% são provavelmente funcionalidades superficiais, responsáveis pela maior parte do custo de se desenvolver o software – em outras palavras, desperdício. Além disso, ainda aumentam significativamente a complexidade da base do código e o custo para mantê-lo.

O custo da complexidade é exponencial, não linear. Quando a base de código é muito complexa, a confiança de realizar alterações de forma segura é pequena. Consequentemente, equipes inteligentes mantêm seu código fonte simples, claro e reduzido. Porém, para que isso aconteça, toda nova funcionalidade deve ter uma boa justificativa do ponto de vista do negócio, de forma que seja algo que realmente agregue valor. Poppendieck alerta que software é complexo por natureza, e por isso, não gerenciar sua complexidade cuidadosamente é uma receita para o fracasso.

Desperdício com documentação

Produzir documentos que ninguém lê é desperdício. Documentar é uma atividade que consome o tempo das pessoas e aumenta o tempo de desenvolvimento.

Evite documentações burocráticas que não serão lidas por ninguém e não agregarão valor ao produto, e concentre-se em documentar somente o que for necessário.

Cuidado, porém, para não levar essa ideia ao extremo. Não deixe de documentar o que for importante.

Larry Burns [16] dá algumas dicas para evitar desperdícios com documentação:

- Não produza documentação que não será lida.
- Evite produzir documentação que não será atualizada (a menos que seja deliberadamente produzida para satisfazer um requisito temporário, como atender uma licitação, por exemplo).
- Não documente coisas óbvias, triviais, redundantes ou informações que podem facilmente ser obtidas em outras fontes.
- A documentação deve estar disponível e poder ser facilmente acessada pelos interessados.

É importante encontrar um equilíbrio saudável, e discernir entre o que deve e o que não deve ser documentado.

Desperdício por falta de foco

De acordo com o livro Peopleware, de Tom DeMarco e Timothy Lister, toda vez que alguém é interrompido durante a execução de uma tarefa para trabalhar em outra, ou ajudar alguém em algum assunto de contexto diferente do que estava fazendo, um tempo significativo é gasto para se concentrar na nova tarefa e voltar ao estado de produtividade anterior [39]. Logo, quanto mais interrupções houver, maior será o desperdício. Por essa mesma razão não é recomendável que um desenvolvedor pertença a várias equipes.

A programação em par é um bom exemplo de técnica que ajuda a evitar interrupções, uma vez que duas pessoas trabalhando juntas e trocando conhecimentos muito provavelmente não precisarão interromper outros membros do time para resolver problemas.

Desperdício por atrasos

Uma das maiores fontes de desperdício no desenvolvimento de software são os atrasos. Atrasos por aprovação ou revisão de requisitos, atrasos por contratação de pessoal, atrasos nos testes, atrasos na entrega etc.

Quanto maior o atraso, mais tempo será necessário para que o cliente obtenha retorno sobre o investimento realizado no desenvolvimento do software.

As técnicas estudadas anteriormente para limitar o trabalho em progresso, e entregas frequentes forçam o processo a ser mais enxuto e fluído reduzindo assim esse tipo de desperdício.

Desperdício por recursos ou pessoas indisponíveis

Quando o *Product Owner* não está presente e não há ninguém que o represente, e existe alguma dúvida, ponto a ser esclarecido ou decisão a ser tomada, é preciso esperar que ele esteja presente para dar continuidade ao desenvolvimento, ou no pior dos casos, por falta de informações, o desenvolvedor pode seguir por um caminho inadequado que provavelmente resultará em um defeito.

Por essa razão é muito importante que o *Product Owner* esteja presente, próximo da equipe.

Desperdício por defeitos

Defeitos são sem dúvida uma das maiores fontes de desperdício no desenvolvimento de software, e quanto mais tempo um defeito leva para ser identificado, maior será o desperdício gerado.

Por essa razão, defeitos devem ser identificados e resolvidos o mais rápido possível.

Essas são apenas algumas das muitas fontes de desperdícios que todas as equipes de desenvolvimento de software possuem em maior ou menor grau. O exercício constante de analisar e questionar o processo é essencial para que eles possam se tornar cada vez menores e menos relevantes.

5.6 E agora, o que eu faço amanhã?

Quais métricas vocês utilizam para acompanhar a produtividade e qualidade do trabalho da sua equipe? Revise as métricas apresentadas na seção 5.2 e verifique se alguma delas poderia ajudar.

Que tal utilizar um dos gráficos apresentados (CFD, Burndown ou Burnup) e deixá-lo visível para que toda a equipe possa acompanhar o progresso da iteração?

Quais dos desperdícios apresentados na seção 5.5 você reconhece no seu projeto? O que você poderia fazer para ajudar a reduzir ou eliminá-lo?

Procure uma técnica de retrospectiva diferente, uma que nunca foi utilizada em sua equipe antes, e ofereça-se para ser o facilitador da próxima retrospectiva.

Capítulo 6

Otimizando o Sistema

Esse é o quarto nível de fluência, falaremos sobre o alinhamento da organização como um todo, os valores ágeis e práticas ágeis, assim como os objetivos da organização, e de como a agilidade pode impactar de forma sistemática a performance da organização além do time.

A teoria do pensamento sistêmico nos ensina que um sistema consiste de partes interdependentes que interagem entre si por um mesmo propósito. Um sistema não é apenas a soma de suas partes, mas também o produto de suas interações.

As melhores partes não necessariamente fazem o melhor sistema, de forma que a habilidade de um sistema de atingir o seu propósito depende de quão bem suas partes trabalham juntas e não apenas de como atuam individualmente.

Em projetos de desenvolvimento de software, é notável que exista uma tendência de se medir o progresso do projeto através de três métricas: escopo, prazo e custo.

A contrassenso, mesmo otimizando cada uma dessas métricas, nem sempre o projeto será bem sucedido, pois elas não levam a qualidade e nem mesmo a satisfação do cliente em consideração. Para não ser pego de surpresa, Tom e Mary Poppendieck

sugerem aplicar uma métrica de mais alto nível, que considere o resultado de forma mais ampla: o retorno sobre o investimento. E complementam: "Se você otimizar o que realmente importa, os outros números tomarão conta de si mesmos" [48]. Por isso devemos dar preferência para métricas globais em vez de métricas locais [8].

Para melhor ilustrar esse conceito, imagine que determinada empresa terceiriza os testes dos softwares que desenvolve, e que a empresa terceirizada recebe por defeito encontrado no sistema. Espera-se então que todos os defeitos sejam identificados pela empresa terceirizada antes que o software entre em produção. Entretanto, quando um defeito for encontrado, provavelmente a empresa terceira não irá colaborar devidamente com a equipe que desenvolveu o software para que defeitos de mesma natureza não voltem a ocorrer porque não é de seu interesse que a quantidade de defeitos diminua.

Se os defeitos forem encontrados e resolvidos antes do software ir para produção, então, a métrica deve apresentar um bom resultado e pode-se vir a pensar que as coisas estão indo bem, mas na verdade não está se considerando a real causa do defeito, e o problema não está sendo resolvido em sua raiz.

O objetivo é otimizar o processo da cadeia de valor como um todo e não em cada etapa do processo isoladamente.

6.1 A Gestão pode ser Ágil?

"Aqueles que não fazem acontecer não são nem gerentes nem líderes."
– Johanna Rothman e Esther Derby

Muitas pessoas acreditam que a gestão não importa, e que bons técnicos vão produzir bons resultados, independente da qualidade da gestão. Johanna Rothman e Esther Derby discordam disso. Para elas os bons gestores atingem metas e desenvolvem pessoas. Elas afirmam ainda acreditarem que muitos gerentes querem ser bons gerentes, porém, não sabem como fazer um bom trabalho.

Se você não está certo de quem faz o papel de gestor da sua organização, pergunte sobre quem se responsabiliza pelo treinamento e desenvolvimento da carreira das pessoas. Quem dá feedback sobre o trabalho das pessoas? Quem monitora o trabalho da equipe de forma sistêmica? Essa pessoa provavelmente assumiu o papel de gestor. O ponto é que a gestão está sempre presente, a questão é se o papel de gestor está dividido entre todas as pessoas envolvidas ou se está representado por alguém.

Jurgen Appelo, no Livro Management 3.0, propõe uma nova abordagem para a gestão, mais alinhada aos métodos ágeis, e coerente com a teoria da complexidade.

"Para todo problema complexo há uma resposta clara, simples e errada".
– H.L. Mencken, Jornalista e Escritor (1880–1956)

O mundo mudou, e muitos dos conceitos que aplicamos em nossa gestão moderna foram herdados da era industrial e pré-industrial quando lidamos com profissionais de perfis completamente diferentes de nossos atuais profissionais da era do conhecimento. Devemos reinventar a gestão para que possamos endereçar os desafios que o mundo dos negócios nos oferece nos tempos atuais.

Muitos métodos e filosofias como Six Sigma, Total Quality e Teoria das Restrições foram muito importantes para melhorar a gestão de nossas organizações, porém estes ainda partem do mesmo princípio de que as organizações operam de cima para baixo através de uma hierarquia. A gestão 3.0 trás uma abordagem sistêmica que trata a organização como sistema vivo que está constantemente em processo de aprendizagem e adaptação.

A grande dificuldade que temos em lidar com sistemas complexos se deve às nossas mentes lineares que preferem causalidade à complexidade. Nós procuramos propósito e causa em todas as coisas, buscamos por causa e efeito em tudo. Estamos acostumados a estudar e aprender de forma linear, a contar histórias, e é dessa forma que enxergamos o mundo, como um lugar repleto de eventos fáceis de serem explicados através de simples efeitos e causas. Gerald Weinberg chamou isso de Falácia da Causalidade.

A gestão ágil tem suas bases na complexidade, por isso, o foco da gestão ágil deve estar na adaptabilidade ao invés de previsibilidade, uma vez que se aceita que muitos fatores em projetos são imprevisíveis.

Para entender melhor a gestão 3.0, vamos entender melhor o que é a Gestão 1.0 e a 2.0, a final de contas, esse 3.0 não poderia ser apenas Marketing.

Gestão 1.0: Focada em hierarquias, esta versão da gestão ficou mundialmente conhecida através do modelo comando-e-controle. Poder na mão de poucos em uma estrutura de decisões top-down. Aqui foi criado aquele conhecido jargão "manda quem pode, obedece quem tem juízo". Podemos dizer, sem possibilidade de erro, que ainda é o modelo de gestão mais utilizado "na prática".

Gestão 2.0: Focada em técnicas, esta versão procura "corrigir" alguns dos "problemas" da primeira versão de gestão, mas mantendo a mesma estrutura top-down, o que – não surpreendentemente – resolve muito pouco. Poderíamos dizer que é uma Gestão 1.0 "turbinada" por Balanced Scorecards, Six Sigma, TQM e outros add-ons.

Gestão 3.0: Focada na complexidade, esta gestão encara as organizações como

redes, e não como hierarquias; e nessas redes as pessoas e seus relacionamentos devem estar no foco da gestão, mais do que os departamentos e seus lucros.

A Gestão 3.0 sugere que a gestão trabalhe com seis visões. Não à toa, essas visões são representadas por um modelo um pouco diferente dos conhecidos círculos, flechas e retângulos de desenhos organizacionais (veja na figura 6.1); algo bem mais próximo do que muitas das organizações se parecem: monstros.
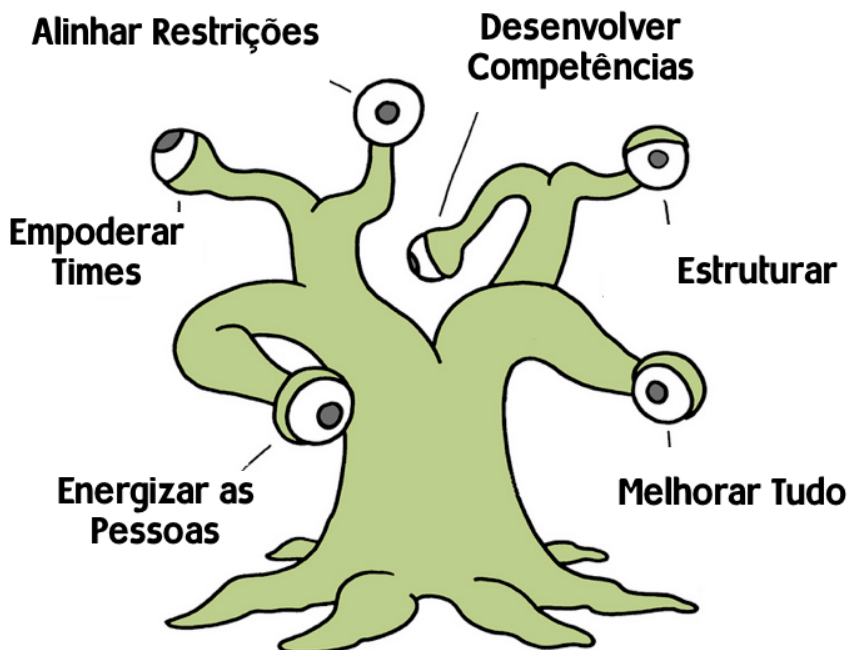
Figura 6.1: Martie, O Modelo de Gestão 3.0 (Desenho por Jurgen Appelo)

Energizar pessoas

"Uma pessoa com paixão é melhor que quarenta pessoas simplesmente interessadas." E – .M. Forster – Romancista Inglês

Pessoas são a parte mais importante de uma empresa, certo? Gerentes devem portanto fazer o que for possível para mantê-las ativas, criativas e motivadas.

A Teoria X da Motivação nos diz que as pessoas não gostam de trabalhar, trabalham por necessidade, porque precisam e por isso, precisam da motivação extrínseca

para trabalhar, e além disso precisam de controle para garantir que estão de fato realizando seu trabalho como devem. Por outro lado, a Teoria Y da Motivação, nos diz que as pessoas gostam de seu trabalho e que encontram prazer em trabalhar, e por isso podem ser motivadas por seus desejos intrínsecos e não somente por algo externo.

A motivação extrínseca, segundo Jurgen, pode ser vista como um fator higiênico, porque podemos compará-la a escovar os dentes, ou tomar banho, por exemplo. Não é a quantidade vezes que escovamos os dentes que nos torna mais feliz, mas é a falta da oportunidade de escovar os dentes que nos faria tristes.

Pesquisas apontaram que a motivação extrínseca (em geral dinheiro, bônus, prêmios, e recompensas) não é suficiente para manter as pessoas motivadas, é preciso entender quais são os desejos e necessidades de cada indivíduo através da abordagem da motivação intrínseca para que então, de alguma forma, seja possível alinhar esses desejos das pessoas à realidade da organização, encontrar uma intersecção entre os propósitos de cada pessoa e o propósito da organização.

É importante notar que pessoas diferentes são motivadas por desejos intrínsecos diferentes, tais como competência, aceitação, curiosidade, honra, idealismo e propósito, independência e autonomia, ordem, poder, status social, relacionados sociais etc. Por isso os gestores precisam estar atentos e devem dedicar-se a conhecer a cada pessoa individualmente, para entender como pode motivá-las.

Empoderar times

Times devem se auto-organizar, mas para isso precisam empoderamento, autorização e confiança da gestão.

Apesar de os gestores ainda deterem o poder de contratar e demitir pessoas, em ambientes de trabalho que em conhecimento é algo crucial, os trabalhadores do conhecimento possuem os trabalhos geralmente críticos, e nesse cenário o gestor pode ser visto como um líder facilitador que empodera os profissionais para que possam tomar as decisões necessárias para realizar o trabalho que são bons em realizar.

Delegar é ato de transferir responsabilidades para outra pessoa, geralmente mantendo-se responsável pelo resultado. Empoderar é mais que delegar — ou dizem alguns, delargar —, para empoderar é preciso considerar o desenvolvimento das pessoas, assumir riscos, e em muitas organizações é preciso mudar a cultura. A teoria da complexidade nos diz que os problemas devem ser resolvidos o mais próximo possível do nível em que eles acontecem.

Empoderar não é algo binário, tratar como dessa forma aliena as outras pessoas

que consideram não ter autoridade de se responsabilizar pelos problemas da organizações, já que "não é minha responsabilidade". Devemos escolher o nível certo de empoderamento.

1) Contar (Nível 1): Você (o gerente) toma a decisão e anuncia às pessoas.
2) Vender (Nível 2): Você toma a decisão mas tenta ganhar o compromisso das pessoas "vendendo" sua ideia para as pessoas.
3) Consultar (Nível 3): Você ouve as opiniões das pessoas e depois toma uma decisão.
4) Concordar (Nível 4): Você convida as pessoas para uma discussão e entra em um acordo com elas, chega a um consenso. Neste nível, sua voz é igual a dos outros.
5) Aconselhar (Nível 5): Você tenta influenciar as pessoas dizendo a elas sua opinião, mas deixa a elas a responsabilidade de tomar uma decisão.
6) Questionar (Nível 6): Você permite que as pessoas decidam antes, mas depois pede a elas que vendam a decisão a você.
7) Delegar (Nível 7): Você permite que as pessoas tomem uma decisão sem qualquer nível de envolvimento da sua parte.

Além de definir os níveis de empoderação, é importante também deixar explícito para que todos saibam "onde podem pisar". Jurgen Appelo desenvolveu uma forma simples e eficiente de tornar a delegação transparente através do Quadro de Delegação [12] (veja na figura 6.2.

Figura 6.2: Quadro da Delegação

No topo do quadro, temos os 7 níveis de emporamento, e nas linhas temos as áreas de delegação-chave (essas áreas vão variar de acordo com o contexto, esse é apenas um exemplo). Nas colunas nos temos a definição de quem é responsável (representado por uma pessoa, um papel, ou uma equipe).

Por exemplo, no quadro da figura 6.2, salário está no nível 2 e na responsabilidade de uma pessoa chamada Bill. Isso quer dizer que Bill define os salários mas tenta "vender" para o time suas decisões sobre salários. Já o item comprar licenças está na responsabilidade da Mary no nível 3, isso significa que apesar de ela ser responsável por tomar essa decisão, ela precisará pedir a opinião do time antes de decidir. Já o item definir tecnologia está na responsabilidade do time no nível 7, o que significa que o time tem total autonomia para decidir sobre tecnologia e que não há necessidade de consultar a gestão antes de decidir.

O quadro de autoridade é uma ótima forma para tornar transparente quem pode fazer o quê. Identifique quais são as áreas de decisão-chave na sua organização e procure definir os níveis de delegação de cada um. Uma vez visível e transparente, será mais fácil de questionar e identificar o que pode mudar para melhor.

Alinhar Restrições

"O que é medido é gerido"
– Peter Drucker

A Gestão 3.0 nos apresenta o empoderamento como um investimento que fazemos nas pessoas. É preciso compreender que será preciso tempo para que as pessoas

ganhem experiência e adquiram conhecimento para que possam fazer determinadas coisas tão bem quanto quem as delegou é capaz de fazer.

Alinhar restrições: auto-organização pode não funcionar, para diminuir esta possibilidade dê às pessoas um propósito claro e metas compartilhadas.

A auto-organização pode levar a qualquer resultado, bom ou ruim, faz parte do papel do gestor dar às pessoas uma meta compartilhada para que possam se auto-organizar, de uma forma que produza valor para a organização. O objetivo desta meta compartilhada é dar às pessoas uma direção a seguir.

Nos últimos anos, temos repetido que nossas metas devem ser SMART (Específicas, Mensuráveis, Atingíveis, Realistas, e Oportunas), porém do ponto de vista da complexidade, podemos entender que metas com objetivos específicos possuem característica diferente umas das outras. Por exemplo, se a sua meta é aproveitar as férias na Noruega, como você vai medir isso? Contará seu número de sorrisos por dia?

Além disso é necessário ter métricas diferentes para os diferentes interessados do projeto (equipe, cliente, gerente, comunidade etc.). Os resultados devem ser explícitos e visíveis para que os envolvidos possam entender como estão e aonde devem chegar e tomar ações para melhorar.

Desenvolver competências

O encontro da preparação com a oportunidade gera o rebento que chamamos sorte.
– Anthony Robbins

Times não conseguirão atingir suas metas caso alguns de seus membros não estiverem suficientemente capacitados. A gestão deve contribuir para o desenvolvimento das competências das pessoas.

Mais adiante nesse capítulo trataremos de várias práticas úteis que podem ser utilizadas pelo time, com apoio da Gestão para fomentar o aprendizado e capacitação das pessoas.

Estruturar

Muitos times trabalham dentro de um contexto organizacional complexo, portanto é importante considerar estruturas que promovam a comunicação. Como dividir as pessoas em equipes? Unir as pessoas de acordo com suas funções ou formar equipes que entreguem o produto do início ao fim? Como essas equipes poderão se

comunicar, diretamente ou por intermédio de alguém? Qual a quantidade ideal de membros em um time ágil? Todas essas são questões relevantes em se tratando de estruturar a organização.

Mais adiante, falaremos de formações de equipes, e abordaremos esse assunto com mais profundidade.

Melhorar tudo

O importante é não parar de questionar.
– Albert Einstein

Enquanto muitos de nós passa a vida reclamando que as coisas não são da forma como gostaríamos que fossem (que os políticos são corruptos, que a empresa é chata, que o mercado está dominado por mercenários, ou seja lá o que for que não está bem), Mahatma Gandhi foi um verdadeiro exemplo deixando-nos essa filosofia: "Seja a mudança que você quer ver no mundo".

Agilidade tem a ver com Melhoria Contínua. E nada melhora se permanecer como está. As coisas melhoram quando mudam. Então, para melhorar, pessoas, times, e organizações precisam melhorar continuamente para evitar falhas e evoluir sempre.

A Gestão 3.0 traz uma abordagem sistêmica para Gestão 3.0 que consiste em [10]:

1) Considerar o sistema: Uma rede social é complexa e adaptativa. Ela vai se adaptar às suas ações, e você também deve se adaptar à rede social. É como dançar com o sistema.

2) Considerar os indivíduos: Saiba que as pessoas são a parte crucial do sistema social, e que elas são diferentes umas das outras. Não há uma abordagem única que sirva para todos. Simplesmente pedir para que as pessoas mudem, raramente, é suficiente. A diversidade é o que faz sistemas complexos funcionarem, e é por isso que trabalhar com uma diversidade de métodos é crucial para lidar com as pessoas.

3) Considerar as interações: Um comportamento se espalha através de um sistema complexo. Em uma rede social, tudo gira em torno de indivíduos e das interações entre eles. Comportamentos se espalham de forma viral, e estimular a rede social pode ajudar a se vencer a resistência às mudanças e a transformar uma organização por completo.

4) Considerar o ambiente: As pessoas sempre se organizam dentro do contexto de um ambiente. O ambiente determina como o sistema pode se auto-organizar, e você deve ser capaz de ajustar o ambiente. Isso porque o comportamento das pessoas depende do ambiente, e se você mudar o ambiente, você também muda as pessoas.

No livro Como Mudar o Mundo: Gestão de Mudanças 3.0, Jurgen Appelo, descreve em detalhes como você pode aplicar cada uma dessas abordagens para se tornar um agente de mudanças bem sucedido.

6.2 Feedback

Não é à toa que um dos valores de XP é Feedback [13]. A constante mudança do mundo à nossa volta cria a necessidade de darmos e receber feedback com frequência.

Desenvolvedores precisam receber feedback sobre as alterações que fizeram no software, precisam receber feedback a cada alteração para que saibam se os testes de regressão continuam passando, por isso utilizam integração contínua e TDD. Além disso, precisam saber se o código que escrevem está legível do ponto de vista de seus pares. O *Product Owner* precisa de feedback dos usuários, o time precisa de feedback da gestão, e a gestão de feedback do time. Os ciclos de feedback são importantes e necessários em diversas atividades e relacionamentos existentes no processo de desenvolvimento.

Quanto mais rápido e frequente for o feedback, mais rápido pode-se responder, corrigir e melhorar. Existem algumas práticas e ferramentas que podem ajudar as pessoas a dar e receber feedback. A seguir, estudaremos algumas delas.

Reuniões One-on-ones: conhecendo o time

Para liderar pessoas, é preciso conhecê-las bem. Cada pessoa é única e possuir diferentes desejos, pontos fortes e fracos. Um bom gestor precisará saber quem são as pessoas (seus pontos fortes, fracos e interesses) e em que estão trabalhando, qual a missão da equipe e de que forma ela agrega valor e como as equipes se encaixam na organização como um todo.

As reuniões one-on-ones basicamente são oportunidades em que o gestor se reúne com cada membro da equipe, um a um para conversar, e perguntar coisas como:

- Como as coisas estão?
- Como seu projeto está indo?
- O que você mais gosta no seu trabalho?
- Quais são os aspectos mais frustrantes?
- Você poderia me falar mais a respeito de ...?

Reuniões One-on-ones, realizadas da forma correta, constroem bons relacionamentos. Gestores que fazem essas reuniões com frequência muitas vezes as consideram um dos melhores usos de seu tempo e uma ferramenta primordial. As one-on-ones podem ajudar sua equipe a saber o que você espera deles, além de criar um canal para coaching, desenvolvimento de carreira, confiança, feedback e oportunidade para reportar o status do trabalho.

Tothman e Derby recomendam que os gerentes façam reuniões one-on-ones semanalmente. Na Bluesoft nós as fazemos a cada três meses, e sem dúvidas, podemos dizer que elas trazem bons resultados.

Dicas para Reuniões One-on-ones

- Ritmo: Encontre as pessoas no mesmo dia e por volta do mesmo horário, isso cria ritmo e faz com que as pessoas venham preparadas.
- Presença: não permita que a reunião seja interrompida por ligações, e-mails etc.
- Compromisso: cancelar ou desmarcar por qualquer razão demonstra que você não se importa.
- Consistência: Procure manter um formato semelhante em diferentes encontros.
- Adaptabilidade: Adapte a frequência, formato ao momento e circunstancias do time.

Não se engane com a ideia de que o gestor não deveria passar tanto tempo com as pessoas para não perder o foco da gestão, porque isso faz parte da gestão. Aprender sobre as pessoas faz parte. Saber em que elas boas e quais são seus pontos a melhorar faz parte. Aprender sobre seus desejos e aspirações faz parte.

Feedback 360

Feedback é essencial para o amadurecimento e evolução das pessoas. Precisamos saber se estamos indo bem, e em que precisamos melhorar.

Imagine se você pudesse receber feedback de diversos pontos de vista variados de diferentes pessoas para saber exatamente em que você pode melhorar. É possível! Há uma ferramenta conhecida como Feedback 360 graus.

O feedback pode ser contextualizado em competências que o time julgar necessárias para que possam atingir suas metas, entregar resultados de qualidade, e atender as expectativas dos clientes, e manter um bom relacionamento e ambiente de trabalho. Pode se avaliar tanto competências como comunicação, colaboração, como coisas mas específicas como conhecimentos em tecnologias, ou boas práticas de desenvolvimento.

Como toda ferramenta, pode ser bem ou mal utilizada. Infelizmente, alguns gerentes a utilizam para realizar avaliações tradicionais no estilo de cima para baixo [8] em que apenas gerentes avaliam subordinados e subordinados avaliam gerentes. É importante evitar fazer um relação direta do resultado do feedbacks, com promoções, demissões, punições, bônus etc. para não perder a confiança das pessoas e criar um clima de tensão na equipe.

É importante preparar as pessoas, e explicar o objetivo do feedback, para que não se torne uma sessão de "fogo para todo lado" em que críticas destrutivas são trocadas pelos membros da equipe resultando apenas em egos feridos.

A essência do feedback 360 graus, de um ponto de vista ágil, é que você possa dar feedback para todos os membros do seu time, receber feedback de todos e, em alguns casos, inclusive, se autoavaliar com objetivo de que todos possam utilizar esse feedback para melhorar. Já a forma pode variar bastante. Há diversas abordagens que podem ser usadas, por exemplo, pode ser anônimo, pode ser face a face, pode ser mais subjetivo e aberto, ou mais objetivo e relacionado a metas específicas, o resultado pode ser particular, ou pode ser publicado para todos. Alguns dos formatos mais comuns são:

1. Face a face: Uma reunião informal em um ambiente casual, em que cada participante oferece e recebe feedback de cada um dos outros [8]. De acordo com Jurgen

Appelo, reuniões informais funcionam melhor do que avaliações mais tradicionais porque as pessoas podem discutir abertamente e esclarecer dúvidas para entender melhor pedindo exemplos reais.

2. Envelopes: Essa é uma forma que geralmente é um pouco aberta, em que todos os membros do time, recebem um envelope com seus nomes escritos, contento folhas em branco. A equipe, então, senta em formato de círculo, e os envelopes são passados em sentido horário de forma, que cada um dos membros do time, ao pegar o envelope do colega escreve algo positivo sobre ele e algo que ele pode melhorar. Ao final, os envelopes voltam a mão de seus donos com as folhas preenchidas com o feeback. Cada um fica com seus resultados para analisar o que pode melhorar.

3. Formulários: A forma mais tradicional é através de formulários (que podem ser digitais ou em papel) em que cada membro do time avalia objetivamente cada uma das competências de todos os seus colegas, através de uma nota, que pode ser de 1 a 10, por exemplo. Depois de todos terem avaliado, os resultados são consolidados e cada um recebe seus resultados, assim como sua avaliação em relação a média do time, e sua própria autoavaliação.

Defina um timebox para o feeback, para evitar que a sessão se torne cansativa e interminável.

Você pode tentar diversos formatos até que encontre um que funcione melhor em sua equipe, ou variar o formato a cada nova sessão de feedback. O mais importante é ter em mente que o objetivo é fomentar o feedback para que as pessoas melhorem.

6.3 Escalando Ágil com Programas e Portfólios

E quando uma equipe ágil não for suficiente? E quando a organização tiver vários produtos ou produtos grandes demais para uma única equipe ágil? Como escalar a agilidade? Como determinar quem trabalha no quê? Essas são perguntas comuns que nos fazemos ao nos deparar com situações em que precisamos de diversas equipes para atingir os objetivos da organização.

Desenvolvimento de sistemas de larga escala podem ser realizados através de diversos times trabalhando paralelamente em um mesmo ciclo de releases, geralmente com data e qualidade fixos, porém, com escopo variável [37]. Esse nível de desenvolvimento em que há várias equipes trabalhando com um objetivo final comum é o nível de Programa.

Em um cenário como esses, é comum que alguém desempenhe o papel *Product*

Manager, ou que haja uma comunidade de prática de *Product Owners* para coordenar, alinhar e lidar com esses requisitos de mais alto nível, que são organizados e priorizados em um *Program Backlog*.

Para organizar muitos programas, trabalha-se com portfólios, onde se trata dos requisitos de mais alto nível, os épicos. Nesse nível é realizada a gestão de todos os investimentos da organização que serão realizados através de programas e projetos.

Assim teremos uma hierarquia formada por Portifólios, Programas e Projetos, em que cada projeto será desenvolvido por uma pequena equipe ágil.

6.4 Formação das Equipes

Uma premissa importante do desenvolvimento ágil são as equipes cross-funcionais, ou seja equipes que possuem pessoas com todas as habilidades necessárias para entregar as histórias de usuário do início ao fim. Elas são geralmente chamadas de equipes de *feature teams* ou funcionalidades.

É diferente das equipes funcionais que, geralmente, eram formadas nos processos tradicionais, com base na função, por exemplo, equipe de programadores, equipes de analistas, equipes de testadores etc. Em organizações que utilizam esses processos tradicionais geralmente o trabalho passa de uma equipe para a outra até que fique pronto, resultando, muitas vezes, em burocracia e problemas de comunicação. A figura 6.3 ilustra a diferença na formação de equipes funcionais e cross-funcionais:
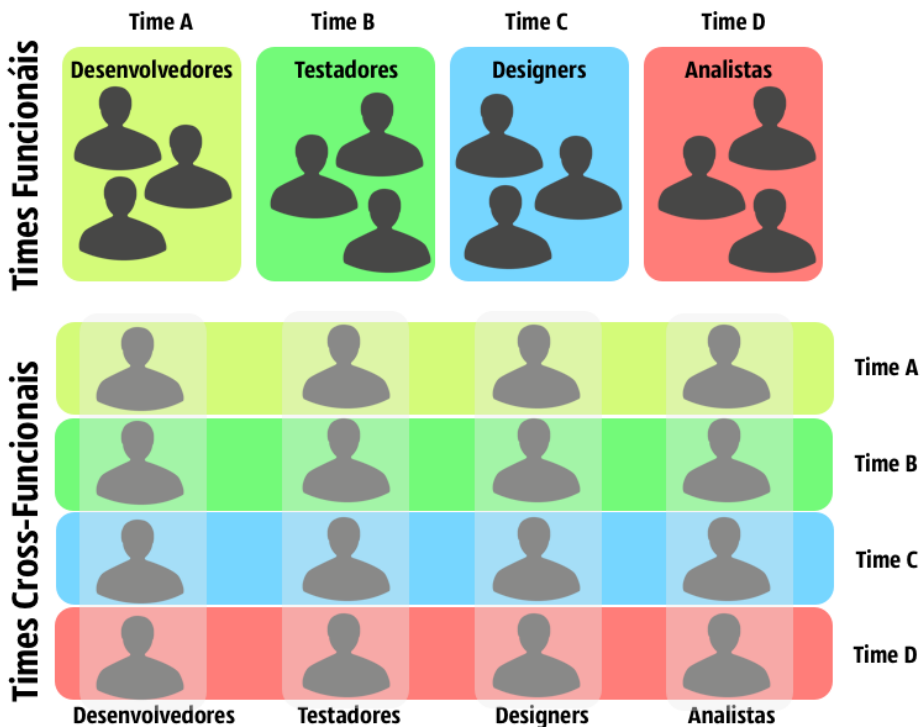
Figura 6.3: Equipes Funcionais e Cross-Funcionais

Nas equipes cross-funcionais todos são responsáveis pelo resultado da entrega, e não apenas por sua área de especialização. Além disso, é comum que os especialistas ensinem os outros um pouco sobre sua área de conhecimento de modo que, com o tempo, todos podem ajudar no que mais for necessário, alavancando assim a produtividade do time [56].

Isso não significa que todos têm que saber fazer todo o trabalho, e que têm que fazer todo o trabalho muito bem. Na verdade, equipes ágeis incentivam a formação de especialistas generalistas, são pessoas que fazem um tipo de trabalho extremamente bem — desenvolver, por exemplo —, porém também são capazes de fazer outros tipo de trabalho — documentar ou testar, por exemplo. Equipes formadas com profissionais assim tendem a eliminar desperdícios e gargalos na organização [8].

Esses profissionais são conhecidos como Profissionais T, porque são especializados em alguma área de conhecimento, representada pela pela "perna" da letra T,

porém são capazes de fazer coisas em outras áreas de conhecimento representado pela amplitude da "cabeça" da letra T, veja um exemplo na figura 6.4:
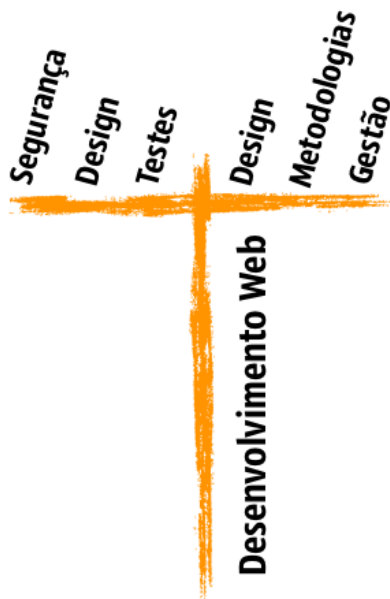
Figura 6.4: Profissional T

Equipes cross-funcionais nem sempre não **feature teams , em podem também ser component teams**, a diferença dessas equipes é que em vez de construir o software de ponta a ponta e entregá-lo pronto para que o cliente final possa utilizar, essas equipes constroem partes da aplicação, ou componentes, que, geralmente, são entregues para que outra equipe possa utilizar e enfim agregar a funcionalidade que precisa para ser entregue ao cliente.

Por exemplo, o time A pode estar trabalhando em um componente visual de front-end, enquanto o time B está preparando o backend da aplicação. Embora seja recomendável trabalhar com **feature teams , Mike Cohn alerta que em certas situações e contextos pode ser interessante se utilizar de component teams**.

Equipes ágeis são capazes de entregar software funcionando a cada iteração porque são formadas de especialistas generalistas capazes de juntos construir funciona-

lidades de ponta a ponta.

Outra diferença notável é o tamanho das equipes, em processos tradicionais era comum encontrar equipes funcionais com mais de 30 pessoas, muitas vezes chegando em extremos de mais de 100. As opiniões quanto a tamanho ideal de uma equipe ágil podem variar, mas, geralmente são equipes pequenas que variam de 5 a 12 integrantes. Quanto mais pessoas envolvidas mais frágil se torna a comunicação do time, mais coordenação se torna necessária, e, geralmente, menor é a produtividade [66].

É importante levar em consideração, como discutimos no capítulo 5 que as pessoas precisam trabalhar juntas por um tempo para que sejam uma equipe de verdade, e não apenas um grupo de trabalho. Por isso, é importante que elas possam permanecer por algum tempo na mesma equipe, embora, de tempos em tempos, pequenas mudanças, rotacionando membros de um equipe para a outra, podem ser positivas para disseminar o conhecimento dentre as equipes.

Contratação Colaborativa

Algo que sem dúvida, influencia na formação de equipes, são os novos membros que são contratados ao longo do tempo. A contratação colaborativa é uma prática que basicamente consiste em permitir que as pessoas da equipe da qual possivelmente o candidato fará parte possa participar do processo de contratação, fazendo perguntas e criando desafios para o candidato.

É claro que essa prática, requer confiança da gestão e maturidade da equipe, mas, dessa forma, além de o time colaborar para selecionar pessoas mais aderentes à cultura e às necessidades da organização, as equipes tendem a aceitar melhor o novo membro, dado que participaram ativamente do processo de contratação e seleção, e concordarão que o novo membro tem as características que consideram importantes para a equipe.

6.5 Práticas de Aprendizagem

"Sejamos claros: Sua carreira é sua responsabilidade, seu empregador não é sua Mãe"
– Robert C. Martin

Há um antigo ditado que diz que "como na natureza, ou você está verde e crescendo ou maduro e apodrecendo". Na área de tecnologia da informação a taxa de mudança e inovação é muito alta, e nós profissionais não podemos parar de aprender.

Jurgen Appelo, afirma que a educação dos colaboradores não é a principal atividade da organização. Por outro lado, esperar que as pessoas se desenvolvam sozinhas nem sempre é uma abordagem bem-sucedida: "Eu acredito que a maioria das pessoas quer aprender porém não sabe como, ou acabam focando em coisas irrelevantes. Defina restrições no ambiente para que as pessoas aprendam coisas relevantes." [11]

Alguns profissionais, infelizmente, não se responsabilizam por seu próprio desenvolvimento profissional, e como em organizações também não há incentivo, esses acabam ficando improdutivos e desatualizados com o tempo.

Em seu livro Clean Code, Robert Martin enfatiza a importância de que todos os profissionais se responsabilizem por suas carreiras [42], algumas organizações incentivam o aprendizado e criam um ambiente repleto de recursos para que as pessoas naturalmente estejam sempre aprendendo algo novo. Isso é fantástico, mas, se sua organização não der espaço para atividades de aprendizagem, não desanime por isso, afinal a responsabilidade ainda é sua.

Nesse tópico abordaremos algumas práticas que você e seu time podem utilizar para estimular o aprendizado de todos.

O ambiente faz diferença

A equação de Lewin, desenvolvida pelo psicologo Kurt Lewin, defende que o comportamento de uma pessoa é uma função de sua personalidade e de seu ambiente [45]. Isso significa que ao mudar o ambiente você pode mudar o comportamento de alguém.

Para ilustrar, imagine que você é um desenvolvedor que acabou de ser contratado por uma nova empresa. Em seu primeiro dia de trabalho, notou que todos os seus colegas têm o hábito de ler e estudar bastante, há livros por toda a parte espalhados nas mesas das pessoas, ao ouvir uma conversa sobre tecnologia no café você por um momento pensou que as pessoas estavam falando em grego: Scala, Big Data, Clojure, Crystal, Dataware house, Business Intelligence, Hadoop, Kernel... Na verdade, era apenas uma conversa normal sobre tecnologia com uma série de palavras-chave que você nunca tinha ouvido falar antes.

Em um ambiente como esse, há uma grande probabilidade de que você chegará em casa e, naturalmente, começará a estudar e buscar equiparação com seus colegas

e, para acompanhá-los, criará o hábito de manter-se sempre atualizado.

Agora, pense num cenário diferente, em que ao chegar em seu primeiro dia de trabalho, você percebe que as pessoas são bem acomodadas, que a tecnologia que há empresa está atualmente utilizando já está ultrapassada há muito tempo e que ninguém parece se importar em estudar ou buscar novas abordagens. Você terá a mesma motivação de estudar ao chegar em casa? Provavelmente não.

O que ocorre que é nós temos uma grande tendência de nos conformarmos ao ambiente em que estamos inseridos.

É por isso que um ambiente que facilita e promove o aprendizado é muito importante para fomentar equipes tecnicamente excelentes. A seguir você será apresentado a algumas práticas para ajudá-lo a criar um ambiente que facilita a aprendizagem.

Comprometa-se com sua carreira

Muitos de nós acredita que ao terminar a faculdade está formado mas, na verdade, especialmente em uma área tão volátil quanto a área de tecnologia da informação, essa formação deve durar a vida inteira para que você continue sendo um profissional eficiente e relevante para o mercado de trabalho.

Quando está na faculdade, você dedica em média 4 horas do seu dia para sua formação, para estudar. Depois que acaba a faculdade, muitos, infelizmente, passam a dedicar zero horas. É importante que você tenha um compromisso consigo mesmo de manter-se em constante estado de aprendizagem e para isso você precisará dedicar algumas horas na sua semana. Não pode dedicar 4 horas como fazia na época na faculdade? Não tem problema. Que você dedique 20 minutos por dia então. Mas faça e verá que isso fará grande diferença na sua carreira.

O Básico

Inglês: Nos tempos globalizados em que vivemos saber falar apenas sua língua nativa já não é mais suficiente. Grande parte do material disponível para aprendizado está em inglês, e o tempo de tradução para o português é, muitas vezes, longo demais. Estude inglês de forma a poder ler documentação de projetos open-source, tutoriais, artigos, e assistir a palestras e apresentações.

Leitura: Crie o hábito de ler. Não apenas livros, mas também blogs e portais de tecnologia (como o InfoQ.com, por exemplo). Fique ligado no que está mudando e nas novas tecnologias que estão surgindo a todo momento.

Esteja sempre lendo um livro. Quando terminar de ler este livro, não espere.

Comece a ler um próximo e não pare nunca. Eu, particularmente, mantenho um Backlog de livros no GoodReads.com, há centenas de livros que quero ler e mantenho uma lista priorizada lá, sempre sei qual o próximo livro que começarei a ler assim que acabar o atual.

Cuidado porém, para não cair no hábito de comprar mais livros do que pode ler, falo por experiência própria. Em certa altura da minha vida, percebi que, mais ainda do que ler, eu gostava de comprar livros. Mantenha sua lista de livros, e vá comprando à medida que for terminando de ler o livros que já tem.

Eventos e Mídia: Participe de eventos e conferências sobre tecnologias e assuntos do seu interesse, troque informações com outros profissionais e construa uma rede de relacionamentos. Hoje em dia, temos uma grande riqueza de material em áudio e vídeo disponíveis online (Screencasts, Podcasts, Vídeos), além disso, muitas conferências gravam suas palestras e disponibilizam online para a comunidade. Use tudo isso a seu favor.

Coding Dojo

Coding Dojo é uma reunião de desenvolvedores com o objetivo de trabalhar em um desafio de programação. Eles se divertem e se engajam para trabalhar na solução do problema ao passo que desenvolvem suas habilidades [2].

A premissa básica é que o aprendizado de habilidades de codificação é um processo contínuo que desenvolvedores devem se dar a oportunidade de codificar com o objetivo de treinar. Não importa qual o nível técnico dos desenvolvedores, todos participam e aprendem juntos.

A primeira ideia de sessões de prática de programação individuais e fora do horário de trabalho foi proposta por Dave Thomas [62] posteriormente Laurent Bossavit, propôs Coding Dojos, com mesmo objetivo, porém com a participação de um grupo de programadores [14].

O ambiente de um coding dojo estimula o aprendizado colaboração, diversão, e é propício para se tentar aplicar novas ideias e boas práticas de programação, como Desenvolvimento Guiado por Testes (TDD), Refatoração, e Programação em par, por exemplo. Além disso, desenvolvedores aprendem muito ao ver as diferentes perspectivas e abordagens de outros desenvolvedores.

Tudo que é preciso para realizar um coding dojo é um computador, um teclado, um projetor, um desafio de programação para ser resolvido e desenvolvedores interessados em participar.

Existem diferentes formatos de Coding Dojos, o **Randori Kata** e o **Prepared Kata**.

No formato Randori, os programadores trabalham juntos na solução do desafio de programação [14], seguindo o seguinte processo:

1) Escolher um desafio de programação: Dica, uma simples busca no Google por "source of problems for coding dojos" trará algumas ideias interessantes como: transformar números romanos em arábicos, escrever números por extenso etc.

2) Um par de desenvolvedores vai até o computador e trabalhar em par no problema, a cada X minutos (geralmente, 7 minutos) o navegador, assume o papel de condutor dando lugar para que a alguém da audiência assuma o papel de navegador, e o condutor anterior volta a plateia, o novo par dá continuidade de onde o par anterior estava. O processo se repete de X em X minutos, até o final do timebox. Os desenvolvedores aplicam desenvolvimento guiado por testes, e enquanto a barra estiver verde, indicando que os testes estão passando, todos da plateia podem dar sugestões ou discutir abordagens, mas quando a barra estiver vermelha (indicando que os testes estão quebrando) apenas o navegador e o condutor podem falar.

3) Uma rápida retrospectiva para que os participantes apontem o que correu bem e o que pode ser melhorado no próximo coding dojo.

Já no formato **Prepared Kata** alguém que já resolveu um desafio de programação anteriormente apresenta a solução de um problema passo a passo desde o ponto zero, utilizando desenvolvimento guiado por testes. Cada passo deve fazer sentido para todos os participantes (audiência), de forma que as pessoas podem interromper a qualquer momento em caso de dúvidas [2].

Dojos são uma excelente ferramenta de aprendizado para desenvolvedores, e organizações podem ceder tempo e/ou espaço para que desenvolvedores possam realizá-los com o intuito de praticar suas habilidades e se tornarem melhores profissionais.

Clubes de Discussão de Livros

Um Clube do Livro é um grupo de pessoas leem o mesmo o livro e se reúnem com determinada frequência para discutir capítulos ou trechos do livro, expressando suas opiniões, aprendizados, coisas que gostaram e não gostaram a respeito dos conceitos

e ideias apresentadas do livro. Geralmente, as pessoas vão lendo o livro à medida que os encontros acontecem. São comum, por exemplo, reuniões semanais, em que, antes de cada reunião lê-se um dos capítulos para discussão.

Numa organização em transição para métodos ágeis, por exemplo, pode ser uma excelente ideia formar um clube do livro, e escolher um livro sobre métodos ágeis para envolver toda a equipe no aprendizado que sem dúvida será muito útil na transição.

Brown Bag Sessions

"Brown Bag Sessions" são reuniões durante a pausa para o almoço em que as pessoas fazem apresentações e compartilham informações. Nos Estados Unidos é comum que as pessoas levem seus lanches para o almoço em sacos de papel pardos (brown bags), como aqueles que as padarias costumam utilizar para pães aqui no Brasil, daí o nome "Brown Bag Sessions".

Durante essas reuniões, geralmente, as pessoas comem enquanto assistem a uma apresentação preparada por um dos membros do time. É uma forma fácil e rápida de se criar uma oportunidade para aprendizado, uma vez que é um horário que na maioria das vezes mais pessoas poderão participar do que antes ou depois do horário de trabalho.

Se você gostaria que sua organização, investisse em um tempo durante o horário de trabalho para esse tipo de atividade de aprendizado, "Brown Bag Sessions" podem ser uma ótima forma de se começar, porque não afeta o tempo de trabalho da equipe. Ao passo que as pessoas participam das reuniões e os resultados se mostram promissores, o apoio da gestão para realização dessas atividades poderá ser mais facilmente conquistado.

6.6 Hackathons

Um Hackathon é um dia de trabalho com foco no aprendizado e autodesenvolvimento através de experimentos e testes de novas ideias.

Uma empresa australiana chamada Atlassian possui um evento realizado a cada três meses chamado **ShipIt Days**. Nesse dia todas as empresas da empresa podem escolher uma de suas ideias para trabalhar, de modo que, ao final de 24 horas, todas devem entregar alguma coisa pronta. Outras empresas como a Facebook e Spotify adotaram abordagens semelhantes [11], e atualmente, essa tem sido uma prática de inovação e aprendizado organizacional muito utilizada em startups e organizações

inovadoras.

Os projetos podem ser feitos individualmente, ou podem ser colaborativos. É comum que no início do Hackathon haja um tempo em que todos se reúnem para apresentar suas ideias e então equipes sejam formadas de maneira auto-organizada.

Muitos desses projetos acabam se tornando produtos reais das empresas que realizam Hackathons. É comum que alguma equipe tenha ideias para os produtos que desenvolvem porém, por alguma razão, não estão no backlog, ou têm baixa prioridade. Essa é uma oportunidade de tentar colocar a ideia em prática e apresentar o resultado ao *Product Owner* e ao time. Talvez, a ideia nunca faça parte do produto, de fato, talvez torne-se uma funcionalidade importante do produto; o que interessa é que o aprendizado trazido da experiência poderá ser, por si só, muito recompensador.

Hackathons são oportunidades para que as pessoas possam aprender, experimentar e inovar. São eventos motivadores que permitem que a equipe quebre a rotina e experimente coisas que em um contexto normal talvez fossem arriscadas ou teriam baixa prioridade.

6.7 Comunidades de Prática

Organizações precisam harmonizar práticas, processos, e ferramentas e diversos times e departamentos, além disso as pessoas precisam compartilhar conhecimento e desenvolverem-se além dos limites das organizações [9]. Para esse objetivo cultivar comunidades de prática pode ser de grande ajuda.

Equipes ágeis, geralmente, são cross-funcionais, ou seja, profissionais com papéis e interesses semelhantes estão distribuídos entre diversos times. Nesse cenário, formar comunidades de práticas para criar oportunidades de compartilhamento e aprendizado para esses profissionais pode alavancar consideravelmente o desenvolvimento das pessoas.

Imagine uma organização com 10 times de Scrum, cada time possui um Scrum Master, e pelo menos dois testadores, alguns designers e alguns desenvolvedores. A criação de comunidades de prática nessa organização permitiria que em momentos oportunos todos os Scrum Masters se reunissem para tratar de assuntos relevantes para todos eles, assim como todos os testadores, todos os designers e todos os desenvolvedores. Veja na figura 6.5 um exemplo:
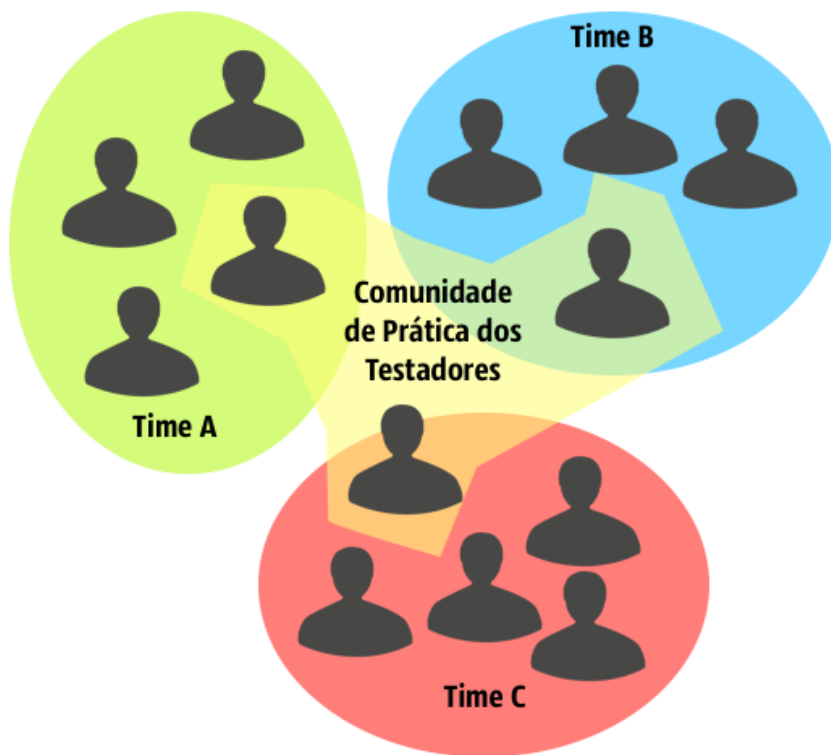
Figura 6.5: Comunidade de Prática

Comunidades de prática são grupos de profissionais que compartilham interesses comuns ou uma área comum de trabalho. Podem ser formadas ao redor de temas, papéis na organização, tecnologias etc. Os participantes aprendem e compartilham informações relevantes, ideias, e boas práticas, e padronizam métodos de trabalho.

A maneira com a qual as organizações lidam com as comunidades de prática pode variar bastante, algumas podem possuir comunidades de práticas informais que não são reconhecidas nem apoiadas pelo organização, outras podem ser reconhecidas e apoiadas [20] e que podem inclusive possuir responsabilidades, e muitas vezes até ter um tempo reservado durante o horário de trabalho para reuniões e discussões. Isso depende do valor que a comunidade de prática pode agregar à organização e da capacidade da gestão de enxergar e apoiar essas iniciativas.

6.8 E agora, o que eu faço amanhã?

Faça um exercício com a sua equipe, crie uma lista com dez problemas que vocês tenham enfrentado na última semana de trabalho, e então para cada um deles verifique como foram resolvidos. Será que vocês realmente foram em busca da causa do problema e o resolveram de uma vez por todas ou simplesmente aplicaram um paliativo para aliviar os sintomas do problema temporariamente?

Que tal marcar uma primeira rodada de one-on-ones?

Procure um assunto de interesse de sua equipe, e proponha a formação de um Clube do Livro para fomentar a discussão e aprendizado.

Marque um primeiro coding dojo com seu time.

Faça uma apresentação sobre esse capítulo, e marque uma "Brown Bag Sessions" para apresentar o que você tem aprendido.

Converse com sua equipe sobre ideias de projetos, melhorias que sempre quiseram ver nos produtos da sua organização, ou experimentos que gostariam de fazer. Que tal propor um Hackathon para criar uma oportunidade de colocar essas ideias em prática?

Existe algum assunto, tecnologia, prática que lhe interessa no contexto da sua organização. Você acha que outras pessoas também poderiam estar interessadas em discutir e evoluir esse tema na sua organização? Que tal formar uma comunidade prática em cima disso?

Proponha à gestão da sua organização a criação de um quadro de autoridade para deixar as autonomia do seu time transparente detre as diversas áreas de decisão-chave do seu projeto.

Capítulo 7

Apêndice A: Ferramentas de Apoio

É essencial para equipes distribuídas, ou equipes que possuem membros que trabalham em regime de *home-office* (trabalhar remotamente de casa), possuir um software que os ajude a manter a visualização do fluxo de trabalho, e mesmo para times que trabalhem colocados, um software pode ser de grande ajuda, embora não seja essencial.

Uma das principais vantagens de se utilizar um software como apoio, são as funcionalidades que permitem armazenar detalhes extras das histórias de usuário, visualização de métricas como tempo de ciclo e *lead time*, geração de gráficos de burndown e burnup, entre outros facilitadores.

Alguns softwares que suportam Kanban são: Acelerato (em português), Lean Kit Kanban, Agile Zen, Target Process, Silver Catalyst, RadTrack, Kanbanery, VersionOne, Greenhopper (com Jira), Flow.io, entre outros.

Se houver um quadro físico e um virtual, é importante garantir que ambos permaneçam atualizados e consistentes. Por isso existe o papel do "Sticky Buddy" (sugerido por Darren Davis), que nada mais é do que alguém que assume a respon-

sabilidade de manter o quadro físico sempre atualizado em caso de alguém estar trabalhando remotamente e não poder mover os itens do quadro físico.

Referências Bibliográficas

[1] Pair programming. http://www.extremeprogramming.org/rules/pair.html.

[2] What is a coding dojo. http://codingdojo.org/cgi-bin/wiki.pl?WhatIsCodingDojo.

[3] Chaos report. http://www.standishgroup.com, 2008.

[4] Lyssa Adkins. *Coaching Agile Teams: A Companion for ScrumMasters, Agile Coaches, and Project Managers in Transition*. Addison-Wesley Professional, 2010.

[5] Marc Benioff; Carlye Adler. *Behind the Cloud: the untold story of how salesforce.com went from idea to billion-dollar company—and revolutionized an industry*. Jossey-Bass, 2009.

[6] Chris Matts; Gojko Adzic. Feature injection: three steps to success. http://www.infoq.com/articles/feature-injection-success, 2011.

[7] Mauricio Aniche. *Test-Driven Development: Teste e Design no Mundo Real*. Casa do Código.

[8] Jurgen Appelo. *Management 3.0: Leading Agile Developers, Developing Agile Leaders*. Addison-Wesley Professional, 2011.

[9] Jurgen Appelo. Business guilds: Management 3.0 workout. http://www.management30.com/workout/business-guilds, 2012.

[10] Jurgen Appelo. *Como Mudar o Mundo: Gestão de Mudanças 3.0*. 2012.

[11] Jurgen Appelo. Exploration days: Management 3.0 workout. http://www.management30.com/workout/exploration-days, 2012.

[12] Jurgen Appelo. Delegation boards: Management 3.0 workout. http://www.management30.com/workout/delegation-boards, 2013.

[13] Kent Beck. *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley Professional, 2004.

[14] Danilo Sato; Hugo Corbucci; Mariana Bravo. Coding dojo: an environment for learning and sharing agile practices. 2008.

[15] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995.

[16] Larry Burns. *Building the Agile Database: How to Build a Successful Application Using Agile Without Sacrificing Data Management*. Technics Publications, 2011.

[17] Alistair Cockburn. *Agile Software Development: The Cooperative Game, Second Edition*. Addison-Wesley Professional, 2006.

[18] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.

[19] Mike Cohn. *Agile Estimating and Planning*. Prentice Hall, 2005.

[20] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009.

[21] Ward Cunningham. Technical debt. http://c2.com/cgi/wiki?TechnicalDebt.

[22] Diana Larsen; Esther Derby. *Agile Retrospectives*. Pragmatic Bookshelf, 2006.

[23] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[24] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[25] Neal Ford. *The Productive Programmer*. O'Reilly Media, Inc., 2008.

[26] Wikimedia Foundation. Missão da wikimedia. http://wikimediafoundation.org/wiki/Mission.

[27] Wikimedia Foundation. Visão da wikimedia. http://wikimediafoundation.org/wiki/Vision.

[28] Kent Beck; Martin Fowler. *Planning Extreme Programming*. Addison-Wesley Professional, 2000.

[29] Martin Fowler. Pair programming misconceptions. http://martinfowler.com/bliki/PairProgrammingMisconceptions.html.

[30] Olav Maassen; Chris Matts; Chris Geary. *Commitment*. Hathaway te Brake Publications, 2013.

[31] Kevlin Henney. *97 Things Every Programmer Should Know*. O'Reilly Media, Inc., 2010.

[32] Norman L. Kerth. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House, 2001.

[33] Laurie Williams; Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.

[34] Henrik Kniberg. *Kanban and Scrum: making the most of both*. Lulu.com, 2010.

[35] Henrik Kniberg. *Lean from the Trenches: Managing Large-Scale Projects with Kanban*. Pragmatic Bookshelf, 2011.

[36] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.

[37] Dean Leffingwell. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise*. Addison-Wesley Professional, 2010.

[38] Scott W. Ambler; Mark Lines. *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press, 2012.

[39] Tom DeMarco; Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, 1999.

[40] Chris Matts; Olav Maassen. Real options underlie agile practices. http://www.infoq.com/articles/real-options-enhance-agility, 2007.

[41] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[42] Robert C. Martin. *The Clean Coder: A Code of Conduct for Professional Programmers*. Prentice Hall, 2011.

[43] Lindsay Ratcliffe; Marc McNeill. *Agile Experience Design: A Digital Designer's Guide to Agile, Lean, and Continuous*. New Riders.

[44] Tim Ottinger. *Continuous Improvement In A Flash: A Guide For Scrum Masters*. Leanpub.

[45] Carol Sansone; Carolyn C Morf; A. T. Panter. *The Sage Handbook of Methods in Social Psychology*. SAGE Publications, 2004.

[46] Roman Pichler. *Agile Product Management with Scrum: Creating Products that Customers Love*. Addison-Wesley Professional, 2010.

[47] Daniel H. Pink. *Drive: The Surprising Truth About What Motivates Us*. Riverhead Books, 2011.

[48] Mary Poppendieck; Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, 2003.

[49] Jonathan Rasmusson. *The Agile Samurai: How Agile Masters Deliver Great Software*. Pragmatic Bookshelf, 2010.

[50] Donald G. Reinertsen. *Managing the Design Factory*. Free Press, 1997.

[51] Ken Howard; Barry Rogers. *Individuals and Interactions: An Agile Guide*. Addison-Wesley Professional, 2011.

[52] Winston Royce. Managing the development of large software systems. 1970.

[53] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Professional, 2012.

[54] Harvard Business School. Hbs elevator pitch builder. http://www.alumni.hbs.edu/careers/pitch.

[55] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004.

[56] Ken Schwaber. *The Enterprise and Scrum*. Microsoft Press, 2007.

[57] Diana Larsen; James Shore;. Your path through agile fluency. http://martinfowler.com/articles/agileFluency.html.

[58] G. R. Stephenson. *Cultural acquisition of a specific learned response among rhesus monkeys. In: Starek, D., R. Schneider, and H. J. Kuhn (eds.). Progress in Primatology*. Fischer, 1967.

[59] Ken Schwaber; Jeff Sutherland. *Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, And Leave Competitors In the Dust*. John Wiley and Sons, 2012.

[60] Vinícius Manhães Teles. *Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade*. Novatec Editora Ltda., 2004.

[61] Andrew Hunt; David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[62] David Thomas. Code kata: How to become a better developer. http://codekata.pragprog.com/2007/01/code_kata_backg.html, 2007.

[63] Alan Shalloway; Guy Beaver; James R. Trott. *Lean-Agile Software Development: Achieving*. Addison-Wesley Professional, 2009.

[64] Kent Beck; Mike Beedle; Arie van Bennekum; Alistair Cockburn; Ward Cunningham; Martin Fowler; James Grenning; Jim Highsmith; Andrew Hunt; Ron Jeffries; Jon Kern; Brian Marick; Robert C. Martin; Steve Mellor; Ken Schwaber; Jeff Sutherland; Dave Thomas. Manifesto for agile software development. http://agilemanifesto.org, 2001.

[65] VersionOne. State of agile development survey results.

[66] James Shore; Shane Warden. *The Art of Agile Development*. O'Reilly, 2007.

[67] Laurie Williams. Strengthening the case for pair-programming.

[68] James Q. Wilson and George Kelling. The police and neighborhood safety. http://www.theatlantic.com/magazine/archive/1982/03/broken-windows/304465/, 1982.